수학 함수들

2015년 10월 14일에 한국어로 옮겨짐
2015년 10월 23일에 마지막으로 갱신됨

원문: https://github.com/torch/torch7/blob/master/doc/maths.md

목차

토치는 텐서 객체를 다루기 위해 매트랩(MATLAB)과 비슷한 함수들을 제공합니다. 그 함수들은 다음과 같은 범주로 분류됩니다.

- zeros, ones 같은 생성자
- diag와 triu 같은 추출자,
- abs와 pow 같은 요소별 수학 연산자,
- BLAS 연산,
- sum과 max 같은 열 또는 행 방향으로 연산,
- trace와 norm 같은 행렬 전체에 대한 연산.
- conv2 같은 컨볼루션과 상호 상관 연산.
- 고윳값/고유벡터 계산 같은 기본적인 선형 대수 연산.
- 텐서에 대한 논리 연산.

기본적으로, 모든 연산들은 결과를 리턴하기 위해 새 텐서 하나를 할당합니다. 또한, 모든 텐서는 타겟 텐서(들)를 첫 번째 인자(들)로 전달할 수 있습니다. 이 경우 그 타겟 텐서(들)는 적절하게 크기가 바뀌어 결과들로 채워질 것입니다. 이 특성은 메모리가 할당되는 과정에 대한 엄격한 통제를 원할 때 특히 유용합니다.

토치 패키지는 객체 지향 문법을 사용하여 텐서 자신에서 한 함수를 바로 호출하는 것과 그 텐서를 선택적 결과 텐서 인자로 전달하는 것이 같도록하는 개념을 채택합니다. 다음 두 호출은 같습니다.

```
torch.log(x,x)
x:log()
```

비슷하게, `torch.conv2` 함수가 다음과 같은 방식으로 사용될 수 있습니다.

```
x = torch.rand(100,100)
k = torch.rand(10,10)
res1 = torch.conv2(x,k)  -- 경우 1

res2 = torch.Tensor()
torch.conv2(res2,x,k)    -- 경우 2
```

```
=res2:dist(res1)
0
```

두 번째 경우의 장점은, 한 루프에서 어떤 새 할당도 없이 같은 `res2` 텐서가 연속적으로 사용될 수 있다는 점입니다.

```
-- 새 메모리 할당 없음...
for i=1,100 do
    torch.conv2(res2,x,k)
end

=res2:dist(res1)
0
```

생성 또는 추출 함수

[res] torch.cat([res,] x_1, x_2, [dimension])

[res] torch.cat([res,] {x_1, x_2, ...}, [dimension])

`x=torch.cat(x_1,x_2,[dimension])`는 차원 `dimension`을 따라 텐서 `x_1`과 `x_2`를 연결한 텐서 `x`를 리턴합니다.

만약 `dimension`이 특정되지 않으면, `dimension`은 마지막 차원입니다.

`x_1`과 `x_2`의 나머지 차원들은 반드시 같아야 합니다.

또한 입력으로 임의의 개수 텐서들을 가진 배열들도 지원합니다.

예:

```
> print(torch.cat(torch.ones(3),torch.zeros(2)))
 1
 1
 1
 0
 0
[torch.Tensor of dimension 5]

> print(torch.cat(torch.ones(3,2),torch.zeros(2,2),1))
 1  1
 1  1
 1  1
```

```
 0  0
 0  0
[torch.DoubleTensor of dimension 5x2]

> print(torch.cat(torch.ones(2,2),torch.zeros(2,2),1))
 1  1
 1  1
 0  0
 0  0
[torch.DoubleTensor of dimension 4x2]

> print(torch.cat(torch.ones(2,2),torch.zeros(2,2),2))
 1  1  0  0
 1  1  0  0
[torch.DoubleTensor of dimension 2x4]

> print(torch.cat(torch.cat(torch.ones(2,2),torch.zeros(2,2),1),torch.rand(3,2),1))
 1.0000  1.0000
 1.0000  1.0000
 0.0000  0.0000
 0.0000  0.0000
 0.3227  0.0493
 0.9161  0.1086
 0.2206  0.7449
[torch.DoubleTensor of dimension 7x2]

> print(torch.cat({torch.ones(2,2), torch.zeros(2,2), torch.rand(3,2)},1))
 1.0000  1.0000
 1.0000  1.0000
 0.0000  0.0000
 0.0000  0.0000
 0.3227  0.0493
 0.9161  0.1086
 0.2206  0.7449
[torch.DoubleTensor of dimension 7x2]
```

[res] torch.diag([res,] x [,k])

`x`가 1차원일 때, `y=torch.diag(x)`는 `x`의 요소들을 대각 요소들로 하는 대각 행렬 하나를 리턴합니다.

`x`가 2차원일 때, `y=torch.diag(x)`는 `x`의 대각 요소들로 구성된 1차원 텐서 하나를 리턴합니다.

`y=torch.diag(x,k)`는 `x`의 `k` 번째 대각 요소들을 리턴합니다, 여기서 `k = 0`은 주 대각이고, `k > 0`은 주 대각보다 위쪽, `k < 0`은 주 대각보다 아래쪽을 가리킵니다.

[res] torch.eye([res,] n [,m])

`y=torch.eye(n)`는 `n x n` 단위 행렬 하나를 리턴합니다.

`y=torch.eye(n,m)`는 `n x m` 단위 행렬 하나를 리턴합니다.

[res] torch.histc([res,] x [,nbins, min_value, max_value])

`y=torch.histc(x)`는 `x`에 있는 요소들의 히스토그램을 리턴합니다. 기본적으로, 그 요소들은 `x`의 최솟값과 최댓값 사이를 100 등분으로 나눈 빈(bin)들로 정렬됩니다.

`y=torch.histc(x,n)`는 위 메소드와 같고 `n` 빈을 가집니다.

`y=torch.histc(x,n,min,max)`는 `n` 빈을 가지고 요소들의 범위로 `[min, max]`를 가진 위 메소드와 같습니다.

[res] torch.linspace([res,] x1, x2, [,n])

`y=torch.linspace(x1,x2)`는 `x1`과 `x2` 사이를 100 등분한 값들을 가진 1차원 텐서 하나를 리턴합니다.

`y=torch.linspace(x1,x2,n)`는 `x1`과 `x2` 사이를 `n` 등분한 값들을 가진 1차원 텐서 하나를 리턴합니다.

[res] torch.logspace([res,] x1, x2, [,n])

`y=torch.logspace(x1,x2)`는 `10^x1`과 `10^x2` 사이를 100 로그적 등분한 값들을 가진 1차원 텐서 하나를 리턴합니다.

`y=torch.logspace(x1,x2,n)`는 `10^x1`과 `10^x2` 사이를 `n` 로그적 등분한 값들을 가진 1차원 텐서 하나를 리턴합니다.

[res] torch.multinomial([res,], p, n, [,replacement])

`y=torch.multinomial(p,n)`는 한 텐서 `y`를 리턴합니다, 한 텐서 `y`의 각 행은 텐서 `p`의 상응하는 행에 위치한 다항 분포에서 샘플된 `n`개의 인덱스들을 담습니다.

`p`의 행들은 더해져 1이 될 필요가 없습니다 (이 경우 우리는 그 값들을 가중치로 사용합니다), 그러나 반드시 음수가 아니어야 하고 더한 값이 0이 아니어야 합니다. 인덱스들은 각각이 언제 샘플되었는지에 따라 왼쪽으로 오른쪽으로 정렬됩니다 (첫 샘플들은 첫 열에 위치됩니다).

만약 `p`가 한 벡터이면, `y`는 한 벡터의 크기 `n`입니다.

만약 `p`가 한 `m` 행 행렬이면, `y`는 한 `m x n` 행렬입니다.

만약 `replacement`가 `true`이면, 샘플들은 **복원추출**됩니다. 만약 `replacement`가 `false`이면, 그것들은 **비복원추출**됩니다. 이것은 한 행을 위한 샘플 인덱스가 뽑힐 때 그것이 그 행을 위해 다시 뽑힐 수 없음을 의미합니다. 이것은 다음과 같은 제약이 있음을 암시합니다. `n`은 반드시 `p` 길이(또는 `p`가 행렬이면 `p`의 열 개수)보다 작아야 함.

`replacement`의 기본값은 `false`입니다.

```
p = torch.Tensor{1, 1, 0.5, 0}
a = torch.multinomial(p, 10000, true)

> a
...
[torch.LongTensor of dimension 10000]

> for i=1,4 do print(a:eq(i):sum()) end
3967
4016
2017
0
```

주의: 만약 이 함수가 결과 텐서를 첫 번째 인자로 넣어 사용된다면(다시 말해, 그 함수의 프로토타입이 `torch.multinomial(res, p, n, [,replacement])`이면), 우리는 그 함수를 다음과 같이 조금 다르게 호출해야만 할 것입니다.

```
p.multinomial(res, p, n, replacement) -- torch.multinomial 대신 p.multinomial
```

이유는 여기서의 결과는 `LongTensor`타입인데, 우리는 `torch.multinomial`을 long 텐서들에 대해 정의하지 않았기 때문입니다.

[res] torch.ones([res,] m [,n...])

`y=torch.ones(n)`는 1로 채워진 크기 `n`의 1차원 텐서 하나를 리턴합니다.

`y=torch.ones(m,n)`은 1로 채워진 `m x n` 텐서 하나를 리턴합니다.

4차원 이상인 경우, 스토리지가 인자로 사용될 수 있습니다. `y=torch.ones(torch.LongStorage{m,n,k,l,o})`.

[res] torch.rand([res,] m [,n...])

`y=torch.rand(n)`는 구간 (0,1)인 한 균등 분포에서 뽑힌 랜덤 수들로 채워진 크기 `n`의 1차원 텐서 하나를 리턴합니다.

`y=torch.rand(m,n)`는 구간 (0,1)인 한 균등 분포에서 뽑힌 랜덤 수들로 채워진 `m x n` 텐서 하나를 리턴합니다.

4차원 이상인 경우, 스토리지가 인자로 사용될 수 있습니다. `y=torch.rand(torch.LongStorage{m,n,k,l,o})`.

[res] torch.randn([res,] m [,n...])

y=torch.randn(n)는 평균이 0이고 분산이 1인 한 정규 분포에서 뽑힌 랜덤 수들로 채워진 크기 n의 1차원 텐서 하나를 리턴합니다.

y=torch.randn(m,n)는 평균이 0이고 분산이 1인 한 정규 분포에서 뽑힌 랜덤 수들로 채워진 m x n 텐서 하나를 리턴합니다.

4차원 이상인 경우, 스토리지가 인자로 사용될 수 있습니다. y=torch.randn(torch.LongStorage{m,n,k,l,o}).

[res] torch.range([res,] x, y [,step])

y=torch.range(x,y)는 크기가 floor((y - x) / step) + 1인 텐서 하나를 리턴합니다. 그 텐서에 있는 값들의 범위는 x부터 y까지이고, 인접한 두 요소 사이 간격은 step(기본값은 1)입니다.

```
> print(torch.range(2,5))
 2
 3
 4
 5
[torch.Tensor of dimension 4]

> print(torch.range(2,5,1.2))
 2.0000
 3.2000
 4.4000
[torch.DoubleTensor of dimension 3]
```

[res] torch.randperm([res,] n)

y=torch.randperm(n)은 1에서 n 사이 정수들의 랜덤 순열 하나를 리턴합니다.

[res] torch.reshape([res,] x, m [,n...])

y=torch.reshape(x,m,n)는 새 m x n 텐서 y를 리턴합니다. y의 요소들은 x에서 행별로 얻어집니다. 그 요소들이 y로 복사됩니다. x의 요소 개수는 반드시 m*n이어야 합니다.

4차원 이상인 경우, 스토리지가 인자로 사용될 수 있습니다.
y=torch.reshape(x,torch.LongStorage{m,n,k,l,o}).

[res] torch.tril([res,] x [,k])

y=torch.tril(x)는 x의 하삼각 부분(lower triangular part)을 리턴합니다. y의 나머지 요소들은 0으로 설정됩니다.

`torch.tril(x,k)`는 `x`의 `k` 번째 대각 요소들과 그보다 아래에 있는 요소들을 0이 아닌 값으로 리턴합니다. `k = 0`은 주 대각 요소들을, `k > 0`은 주 대각 요소들보다 위쪽을, `k < 0`은 주 대각 요소들보다 아래쪽을 가리킵니다.

[res] torch.triu([res,] x, [,k])

`y=torch.triu(x)`는 `x`의 상삼각 부분(upper triangular part)을 리턴합니다. `y`의 나머지 요소들은 0으로 설정됩니다.

`torch.triu(x,k)`는 `x`의 `k` 번째 대각 요소들과 그 위쪽에 있는 요소들을 0이 아닌 값으로 리턴합니다. `k = 0`은 주 대각 요소들을, `k > 0`은 주 대각 요소들보다 위쪽을, `k < 0`은 주 대각 요소들보다 아래쪽을 가리킵니다.

[res] torch.zeros([res,] x)

`y=torch.zeros(n)`는 0으로 채워진 크기가 `n`인 1차원 텐서 하나를 리턴합니다.

`y=torch.zeros(m,n)`은 0으로 채워진 `m x n` 텐서 하나를 리턴합니다.

4차원 이상인 경우, 스토리지가 인자로 사용될 수 있습니다. `y=torch.zeros(torch.LongStorage{m,n,k,l,o})`.

요소별 수학 연산

[res] torch.abs([res,] x)

`y=torch.abs(x)`는 `x`에 있는 요소들의 절댓값들로 구성된 새 텐서 하나를 리턴합니다.

`x:abs()`는 `x`에 있는 각 요소의 절댓값을 계산하여, 그 값을 그 요소가 있던 원래 자리에 다시 저장합니다.

[res] torch.acos([res,] x)

`y=torch.acos(x)`는 `x`에 있는 요소들의 아크코사인 값들들로 구성된 새 텐서 하나를 리턴합니다.

`x:acos()`는 `x`에 있는 각 요소의 아크코사인을 계산하여, 그 값을 그 요소가 있던 원래 자리에 다시 저장합니다.

[res] torch.asin([res,] x)

`y=torch.asin(x)`는 `x`에 있는 요소들의 아크사인 값들들로 구성된 새 텐서 하나를 리턴합니다.

x:asin()는 x에 있는 각 요소의 아크사인을 계산하여, 그 값을 그 요소가 있던 원래 자리에 다시 저장합니다.

[res] torch.atan([res,] x)

y=torch.atan(x)는 x에 있는 요소들의 아크탄젠트 값들들로 구성된 새 텐서 하나를 리턴합니다.

x:atan()는 x에 있는 각 요소의 아크탄젠트를 계산하여, 그 값을 그 요소가 있던 원래 자리에 다시 저장합니다.

[res] torch.ceil([res,] x)

y=torch.ceil(x)는 x에 있는 요소들의 값을 그 값에서 가장 가까운 정수로 올림하여, 그 값들로 구성된 새 텐서 하나를 리턴합니다.

x:ceil()은 x에 있는 요소들의 값들을 그 값에서 가장 가까운 정수로 올림하여, 그 값들을 그 요소들이 있던 원래 자리에 다시 저장합니다.

[res] torch.cos([res,] x)

y=torch.cos(x)는 x에 있는 요소들의 코사인 값들로 구성된 새 텐서 하나를 리턴합니다.

x:cos()는 x에 있는 각 요소의 코사인을 계산하여, 그 값을 그 요소가 있던 원래 자리에 다시 저장합니다.

[res] torch.cosh([res,] x)

y=torch.cosh(x)는 x에 있는 요소들의 쌍곡코사인(hyperbolic cosine) 값들로 구성된 새 텐서 하나를 리턴합니다.

x:cosh()는 x에 있는 각 요소의 쌍곡코사인을 계산하여, 그 값을 그 요소가 있던 원래 자리에 다시 저장합니다.

[res] torch.exp([res,] x)

y=torch.exp(x)는 x에 있는 각 요소에 대한 e(자연 로그)의 x승 값들로 구성된 새 텐서 하나를 리턴합니다.

x:exp()는 x에 있는 각 요소에 대한 e(자연 로그)의 x승 값을 계산하여, 그 값을 그 요소가 있던 원래 자리에 다시 저장합니다.

[res] torch.floor([res,] x)

y=torch.floor(x)는 x에 있는 요소들의 값들을 그 값에서 가장 가까운 정수로 내림하여, 그 값들로 구성된 새 텐서 하나를 리턴합니다.

x:floor()는 x에 있는 요소들의 값들을 그 값에서 가장 가까운 정수로 내림하여, 그 값들을 그 요소들이 있던 원래 자리에 다시 저장합니다.

[res] torch.log([res,] x)

y=torch.log(x)는 x에 있는 요소들의 자연 로그 값들로 구성된 새 텐서 하나를 리턴합니다.

x:log()는 x에 있는 요소들의 자연 로그 값들을 계산하여, 그 값들을 그 요소들이 있던 원래 자리에 다시 저장합니다.

[res] torch.log1p([res,] x)

y=torch.log1p(x)는 x+1에 있는 요소들의 자연 로그 값들로 구성된 새 텐서 하나를 리턴합니다.

x:log1p()는 x+1에 있는 요소들의 자연 로그 값들을 계산하여, 그 값들을 그 요소들이 있던 원래 자리에 다시 저장합니다. 이 함수는 작은 값의 x에 대해 log()보다 더 정확합니다.

x:neg()

x:neg()는 x에 있는 모든 요소의 부호를 반대로 바꿔 그 요소가 원래 있던 자리에 다시 저장합니다.

[res] torch.pow([res,] x, n)

x를 한 텐서, 그리고 n을 한 숫자라고 합시다.

y=torch.pow(x,n)은 x의 n승 값들로 구성된 새 텐서 하나를 리턴합니다.

y=torch.pow(n,x)는 n의 x승 값들로 구성된 새 텐서 하나를 리턴합니다.

x:pow(n)은 x의 n승 값들을 계산하여, 그 값들을 그 요소들이 있던 원래 자리에 다시 저장합니다.

[res] torch.round([res,] x)

y=torch.round(x)는 x에 있는 요소들의 값들을 반올림한 값들로 구성된 새 텐서 하나를 리턴합니다.

x:round()는 x에 있는 요소들의 값들을 반올림하여, 그 값들을 그 요소들이 있던 원래 자리에 다시 저장합니다.

[res] torch.sin([res,] x)

`y=torch.sin(x)`는 `x`에 있는 요소들의 사인 값들로 구성된 새 텐서 하나를 리턴합니다.

`x:sin()`는 `x`에 있는 각 요소의 사인(sine) 값을 계산하여, 그 값을 그 요소가 있던 원래 자리에 다시 저장합니다.

[res] torch.sinh([res,] x)

`y=torch.sinh(x)`는 `x`에 있는 요소들의 쌍곡사인(hyperbolic sine) 값들로 구성된 새 텐서 하나를 리턴합니다.

`x:sinh()`는 `x`에 있는 각 요소의 쌍곡사인 값을 계산하여, 그 값을 그 요소가 있던 원래 자리에 다시 저장합니다.

[res] torch.sqrt([res,] x)

`y=torch.sqrt(x)`는 `x`에 있는 요소들의 제곱근 값들로 구성된 새 텐서 하나를 리턴합니다.

`x:sqrt()`는 `x`에 있는 요소들의 제곱근 값들을 계산하여, 그 값들을 그 요소가 있던 원래 자리에 다시 저장합니다.

[res] torch.tan([res,] x)

`y=torch.tan(x)`는 `x`에 있는 요소들의 탄젠트 값들로 구성된 새 텐서 하나를 리턴합니다.

`x:tan()`는 `x`에 있는 각 요소의 탄젠트 값을 계산하여, 그 값을 그 요소가 있던 원래 자리에 다시 저장합니다.

[res] torch.tanh([res,] x)

`y=torch.tanh(x)`는 `x`에 있는 요소들의 쌍곡탄젠트(hyperbolic tangent) 값들로 구성된 새 텐서 하나를 리턴합니다.

`x:tanh()`는 `x`에 있는 각 요소의 쌍곡탄젠트를 계산하여, 그 값을 그 요소가 있던 원래 자리에 다시 저장합니다.

기본적인 연산

이 절에서, 우리는 텐서를 위한 기본적인 수학 연산들을 설명합니다.

[res] torch.add([res,] tensor, value)

인자로 입력된 값을 텐서에 있는 모든 요소에 더합니다.

`y=torch.add(x,value)`는 새 텐서 하나를 리턴합니다.

`x:add(value)`는 `value`를 더하여, 그 값을 그 요소가 있던 원래 자리에 저장합니다.

[res] torch.add([res,] tensor1, tensor2)

`tensor1`과 `tensor2`를 더하여 그 결과를 `res`에 넣습니다. 두 텐서의 요소 개수는 반드시 같아야 합니다. 그러나 두 텐서의 차원은 다를 수도 있습니다.

```
> x = torch.Tensor(2,2):fill(2)
> y = torch.Tensor(4):fill(3)
> x:add(y)
> = x
 5  5
 5  5
[torch.Tensor of dimension 2x2]
```

`y=torch.add(a,b)`는 새 텐서 하나를 리턴합니다.

`torch.add(y,a,b)`는 `a+b`를 `y`에 넣습니다.

`a:add(b)`는 `b`에 있는 모든 요소들을 (요소별로) `a`로 누적합니다.

`y:add(a,b)`는 `a+b`를 `y`에 넣습니다.

[res] torch.add([res,] tensor1, value, tensor2)

`tensor2`에 있는 요소들에 스칼라 `value`를 곱하고, 거기에 `tensor1`을 더합니다. 두 텐서의 요소 개수는 반드시 같아야 합니다. 그러나 차원은 다를 수도 있습니다.

```
> x = torch.Tensor(2,2):fill(2)
> y = torch.Tensor(4):fill(3)
> x:add(2, y)
> = x
 8  8
 8  8
[torch.Tensor of dimension 2x2]
```

`x:add(value,y)`는 `y`의 요소 값들에 `value`를 곱하여, `x`에 누적합니다.

z:add(x,value,y)는 x + value*y의 결과를 z에 넣습니다.

torch.add(x,value,y)는 x + value*y의 결과를 새 텐서 하나로 리턴합니다.

torch.add(z,x,value,y)는 x + value*y의 결과를 z에 넣습니다.

tensor:csub(value)

인자로 입력된 value를 그 텐서에 있는 모든 요소들에서 빼서 그 요소가 있던 원래 자리에 다리 저장합니다.

tensor1:csub(tensor2)

tensor2를 tensor1에서 뺍니다. 요소 개수는 반드시 같아야합니다. 차원은 다를 수도 있습니다.

```
> x = torch.Tensor(2,2):fill(8)
> y = torch.Tensor(4):fill(3)
> x:csub(y)
> =x
 5 5
 5 5
[torch.Tensor of dimension 2x2]
```

a:csub(b) a - b의 결과를 a로 넣습니다.

[res] torch.mul([res,] tensor1, value)

tensor1 안에 있는 모든 요소들에 value를 곱합니다.

z=torch.mul(x,2)는 x*2의 결과를 새 텐서 하나로 리턴합니다.

torch.mul(z,x,2)는 x*2의 결과를 z에 넣습니다.

x:mul(2)는 x의 모든 각 요소에 2를 곱하여 그 요소가 있던 원래 자리에 다시 저장합니다.

z:mul(x,2)는 x*2의 결과를 z에 넣습니다.

[res] torch.clamp([res,] tensor, min_value, max_value)

tensor의 모든 요소들을 범위 [min_value, max_value]로 줄입니다. 다시 말해,

```
y_i = x_i,       (만약 x_i >= min_value 또는 x_i <= max_value)
    = min_value, (만약 x_i < min_value)
    = max_value, (만약 x_i > max_value)
```

z=torch.clamp(x,0,1)는 값이 0과 1 사이로 제한된 x의 결괏값들로 구성된 새 텐서 하나를 리턴합니다.

torch.clamp(z,x,0,1)는 그 결과를 z로 넣습니다.

x:clamp(0,1)는 clamp 연산을 제자리에서 수행합니다 (즉, 연산 결과를 다시 x에 저장합니다).

z:clamp(x,0,1)는 그 결과를 z에 넣습니다.

[res] torch.cmul([res,] tensor1, tensor2)

tensor1와 tensor2의 요소별 곱셈. 요소 개수는 반드시 같아야 합니다. 차원은 다를 수도 있습니다.

```
> x = torch.Tensor(2,2):fill(2)
> y = torch.Tensor(4):fill(3)
> x:cmul(y)
> = x
 6  6
 6  6
[torch.Tensor of dimension 2x2]
```

z=torch.cmul(x,y)는 새 텐서 하나를 리턴합니다.

torch.cmul(z,x,y)는 그 결과를 z에 넣습니다.

y:cmul(x)는 y의 모든 요소를 그에 상응하는 x의 요소와 곱합니다. 결괏값들은 y에 저장됩니다.

z:cmul(x,y)는 그 결과를 z에 넣습니다.

[res] torch.cpow([res,] tensor1, tensor2)

tensor1의 요소들을 받아서 tensor2의 요소들로 거듭제곱하는 요소별 제곱 연산. 요소 개수는 반드시 같아야 합니다. 그러나 차원은 다를 수도 있습니다.

```
> x = torch.Tensor(2,2):fill(2)
> y = torch.Tensor(4):fill(3)
> x:cpow(y)
> = x
 8  8
```

```
 8  8
[torch.Tensor of dimension 2x2]
```

z=torch.cpow(x,y)는 새 텐서 하나를 리턴합니다.

torch.cpow(z,x,y)는 그 결과를 z에 넣습니다.

y:cpow(x) 는 y의 모든 요소들을 받아서 그에 상응하는 x의 요소들로 제곱합니다.

z:cpow(x,y)는 그 결과를 z에 넣습니다.

[res] torch.addcmul([res,] x [,value], tensor1, tensor2)

tensor1과 tensor2를 요소별로 곱합니다. 그 결과에 스칼라 value(만약 없으면 1)를 곱합니다. 그 결과에 x를 더합니다. 요소 개수는 반드시 같아야 합니다. 그러나 차원은 다를 수도 있습니다.

```
> x = torch.Tensor(2,2):fill(2)
> y = torch.Tensor(4):fill(3)
> z = torch.Tensor(2,2):fill(5)
> x:addcmul(2, y, z)
> = x
 32  32
 32  32
[torch.Tensor of dimension 2x2]
```

z:addcmul(value,x,y)는 그 결과를 z에 누적합니다.

torch.addcmul(z,value,x,y)는 그 결과를 새 텐서 하나로 리턴합니다.

torch.addcmul(z,z,value,x,y)는 그 결과를 z에 넣습니다.

[res] torch.div([res,] tensor, value)

tensor의 모든 요소들을 주어진 value로 나눕니다.

z=torch.div(x,2)는 x/2의 결과들로 구성된 새 텐서 하나를 리턴합니다.

torch.div(z,x,2)는 x/2의 결과를 z에 넣습니다.

x:div(2)는 x의 모든 요소들을 2로 나눈 값을 그 요소가 원래 있던 자리에 저장합니다.

z:div(x,2) 는 x/2의 결과를 z에 넣습니다.

[res] torch.cdiv([res,] tensor1, tensor2)

tensor1의 각 요소를 그에 상응하는 tensor2의 각 요소로 나눕니다 (요소별 나눗셈). 요소 개수는 반드시 같아야 합니다. 그러나 차원은 다를 수도 있습니다.

```
> x = torch.Tensor(2,2):fill(1)
> y = torch.range(1,4)
> x:cdiv(y)
> = x
 1.0000  0.5000
 0.3333  0.2500
[torch.Tensor of dimension 2x2]
```

z=torch.cdiv(x,y)는 새 텐서 하나를 리턴합니다.

torch.cdiv(z,x,y)는 그 결과를 z에 넣습니다.

y:cdiv(x)는 y의 모든 요소들을 그에 상응하는 x의 요소들로 나눕니다.

z:cdiv(x,y)는 그 결과를 z에 넣습니다.

[res] torch.addcdiv([res,] x [,value], tensor1, tensor2)

tensor1의 각 요소를 그에 상응하는 tensor2의 각 요소로 나눕니다 (요소별 나눗셈). 그 결과에 스칼라 value를 곱합니다. 그 결과를 x에 더합니다. 요소 개수는 반드시 같아야 합니다. 그러나 차원은 다를 수도 있습니다.

```
> x = torch.Tensor(2,2):fill(1)
> y = torch.range(1,4)
> z = torch.Tensor(2,2):fill(5)
> x:addcdiv(2, y, z)
> = x
 1.4000  1.8000
 2.2000  2.6000
[torch.Tensor of dimension 2x2]
```

`z:addcdiv(value,x,y)`는 그 결과를 `z`에 누적합니다.

`torch.addcdiv(z,value,x,y)`는 그 결과를 가진 새 텐서 하나를 리턴합니다.

`torch.addcdiv(z,z,value,x,y)`는 그 결과를 `z`에 넣습니다.

[number] torch.dot(tensor1,tensor2)

`tensor1`과 `tensor2`의 내적(dot product)을 수행합니다. 요소 개수는 반드시 같아야 합니다. 두 텐서는 1차원 벡터 하나처럼 보여집니다.

```
> x = torch.Tensor(2,2):fill(2)
> y = torch.Tensor(4):fill(3)
> = x:dot(y)
24
```

`torch.dot(x,y)`와 `x:dot(y)`는 `x`와 `y`의 내적을 리턴합니다.

[res] torch.addmv([res,] [beta,] [v1,] vec1, [v2,] mat, vec2)

`mat`(2차원 텐서)와 `vec2`(1차원 텐서)의 행렬 벡터 곱을 수행합니다. 그리고 그 결과를 `vec1`에 더합니다.

선택적인 값인 `v1`과 `v2`는 스칼라입니다. `v1`과 `v2`는 각각 `vec1`과 `vec2`를 확대합니다.

선택적인 값인 `beta`는 스칼라입니다. `beta`는 결과를 텐서에 누적하기 전에, `res` 텐서의 크기를 조절합니다. 기본값은 1.0입니다.

다른 말로,

```
res = (beta * res) + (v1 * vec1) + (v2 * (mat * vec2))
```

차원(sizes)은 반드시 행렬 곱 연산이 가능하게 설정되어야 합니다. 만약 `mat`가 `n x m` 행렬이면, `vec2`는 반드시 크기 `m`인 벡터여야하고, `vec1`은 반드시 크기 `n`인 벡터여야 합니다.

```
> x = torch.Tensor(3):fill(0)
> M = torch.Tensor(3,2):fill(3)
> y = torch.Tensor(2):fill(2)
> x:addmv(M, y)
> = x
 12
 12
 12
[torch.Tensor of dimension 3]
```

`torch.addmv(x,y,z)`는 그 결과로 구성된 새 텐서 하나를 리턴합니다.

`torch.addmv(r,x,y,z)`는 그 결과를 `r`에 넣습니다.

`x:addmv(y,z)`는 `y*z`를 `x`에 누적합니다.

`r:addmv(x,y,z)`는 `x+y*z`의 결과를 `r`에 넣습니다.

[res] torch.addr([res,] [v1,] mat, [v2,] vec1, vec2)

`vec1`(1차원 텐서)과 `vec2`(1차원 텐서)의 외적(outer product)을 수행합니다.

선택적인 값인 `v1`과 `v2`는 스칼라입니다. `v1`은 `mat`를, `v2`는 `(vec1 [외적] vec2)`를 각각 확대합니다.

다른 말로,

```
res_ij = (v1 * mat_ij) + (v2 * vec1_i * vec2_j)
```

만약 `vec1`이 크기 `n`인 벡터이고 `vec2`가 크기 `m`인 벡터이면, `mat`의 차원은 반드시 `n x m`이어야 합니다.

```
> x = torch.range(1,3)
> y = torch.range(1,2)
> M = torch.Tensor(3,2):zero()
> M:addr(x, y)
 1  2
 2  4
 3  6
[torch.Tensor of dimension 3x2]
```

`torch.addr(M,x,y)`는 그 결과를 새 텐서 하나로 리턴합니다.

`torch.addr(r,M,x,y)`는 그 결과를 `r`에 넣습니다.

`M:addr(x,y)`는 그 결과를 `M`에 넣습니다..

`r:addr(M,x,y)`는 그 결과를 `r`에 넣습니다.

[res] torch.addmm([res,] [beta,] [v1,] M [v2,] mat1, mat2)

`mat1`(2차원 텐서)과 `mat2`(2차원 텐서) 사이의 행렬 행렬 곱을 수행합니다.

선택적인 값 `v1`과 `v2`는 스칼라입니다. `v1`은 `M`을 `v2`는 `(mat1 * mat2)`를 각각 확대합니다.

선택적인 값 `beta`는 스칼라입니다. `beta`는 결과를 텐서에 누적하기 전에 결과 텐서의 크기를 조절합니다. 기본값은 1.0입니다.

다른 말로,

```
res = (res * beta) + (v1 * M) + (v2 * mat1*mat2)
```

만약 `mat1`이 `n x m` 행렬이면, 반드시 `mat2`는 `m x p`행렬이고, `M`은 `n x p`행렬이어야 합니다.

`torch.addmm(M,mat1,mat2)`는 그 결과를 새 텐서 하나로 리턴합니다.

`torch.addmm(r,M,mat1,mat2)`는 그 결과를 `r`에 넣습니다.

`M:addmm(mat1,mat2)`는 그 결과를 `M`에 넣습니다.

`r:addmm(M,mat1,mat2)`는 그 결과를 `r`에 넣습니다.

[res] torch.addbmm([res,] [v1,] M [v2,] batch1, batch2)

`batch1`과 `batch2`에 저장된 행렬들을 묶음(batch) 행렬 행렬 곱하고, 그 모든 행렬 곱 결과들을 한 곳으로 누적합니다.

batch1과 batch2는 반드시 각각 같은 수의 행렬을 가진 3차원 텐서여야 합니다. 만약 batch1이 b x n x m 텐서이고, batch2가 b x m x p 텐서이면, res 는 n x p 텐서일 것입니다.

다른 말로,

```
res = (v1 * M) + (v2 * sum(batch1_i * batch2_i, i=1,b))
```

torch.addbmm(M,x,y)는 그 결과를 새 텐서 하나에 넣습니다.

M:addbmm(x,y)는 필요할 경우 M의 크기를 바꾸어, 그 결과를 M에 넣습니다.

M:addbmm(beta,M2,alpha,x,y)는 필요할 경우 M의 크기를 바꾸어, 그 결과를 M에 넣습니다.

[res] torch.baddbmm([res,] [v1,] M [v2,] batch1, batch2)

묶음 덧셈을 가진 batch1과 batch2에 저장된 행렬들의 묶음(batch) 행렬 행렬 곱.

batch1과 batch2는 반드시 각각 같은 수의 행렬을 가진 3차원 텐서여야 합니다. 만약 batch1이 b x n x m 텐서이고, batch2가 b x m x p 텐서이면, res 는 b x n x p 텐서일 것입니다.

다른 말로,

```
res_i = (v1 * M_i) + (v2 * batch1_i * batch2_i)
```

torch.baddbmm(M,x,y)는 그 결과를 새 텐서 하나에 넣습니다.

M:baddbmm(x,y)는 필요할 경우 M의 크기를 바꾸어, 그 결과를 M에 넣습니다.

M:baddbmm(beta,M2,alpha,x,y)는 필요할 경우 M의 크기를 바꾸어, 그 결과를 M에 넣습니다.

[res] torch.mv([res,] mat, vec)

`mat`와 `vec`의 행렬 벡터 곱. 차원(sizes)은 반드시 행렬 벡터 곱 연산이 가능하도록 설정되어 있어야 합니다. 만약 `mat`가 `n x m` 행렬이면, `vec`는 반드시 크기 `m`인 벡터이고 `res`는 반드시 크기 `n`인 벡터여야 합니다.

`torch.mv(x,y)`는 그 결과를 새 텐서 하나에 넣습니다.

`torch.mv(M,x,y)`는 그 결과를 `M`에 넣습니다.

`M:mv(x,y)`는 그 결과를 `M`에 넣습니다.

[res] torch.mm([res,] mat1, mat2)

`mat1`와 `mat2`의 행렬 행렬 곱. 만약 `mat1`이 `n x m` 행렬이면, 반드시 `mat2`는 `m x p` 행렬이고, `res`는 `n x p` 행렬이어야 합니다.

`torch.mm(x,y)`는 그 결과를 새 텐서 하나에 넣습니다.

`torch.mm(M,x,y)`는 그 결과를 `M`에 넣습니다.

`M:mm(x,y)`는 그 결과를 `M`에 넣습니다.

[res] torch.bmm([res,] batch1, batch2)

`batch1`과 `batch2`에 저장된 행렬들의 묶음(batch) 행렬 행렬 곱. `batch1`과 `batch2`는 반드시 각각 같은 수의 행렬을 가진 3차원 텐서여야 합니다. 만약 `batch1`이 `b x n x m` 텐서이고, `batch2`가 `b x m x p` 텐서이면, `res`는 `b x n x p` 텐서일 것입니다.

`torch.bmm(x,y)`는 그 결과를 새 텐서 하나에 넣습니다.

`torch.bmm(M,x,y)`는 필요할 경우 `M`의 크기를 바꾸어, 그 결과를 `M`에 넣습니다.

`M:bmm(x,y)`는 필요할 경우 `M`의 크기를 바꾸어, 그 결과를 `M`에 넣습니다.

[res] torch.ger([res,] vec1, vec2)

`vec1`과 `vec2`의 외적(outer product). 만약 `vec1`이 크기 `n`인 벡터이고 `vec2`가 크기 `m`인 벡터이면, `res`는 반드시 크기 `n x m`인 행렬이어야 합니다.

`torch.ger(x,y)`는 그 결과를 새 텐서 하나에 넣습니다.

`torch.ger(M,x,y)`는 그 결과를 `M`에 넣습니다.

`M:ger(x,y)`는 그 결과를 `M`에 넣습니다.

오버로드된 연산자

텐서들에 기본적인 수학 연산자들(`+`, `-`, `*`, `/`)을 사용할 수 있습니다. 이 연산자들은 편의를 위해 제공됩니다. 비록 이 연산자들이 쓰기 편하긴 하지만, 이 연산자들은 연산 결과를 새 텐서에 담아 리턴합니다. 따라서 수학 연산자들을 사용한 연산들은 이전 절의 연산들만큼 빠르지 않습니다.

유념할 또다른 중요한 점은 이 연산들이 오직 첫 피연산자가 텐서일 때만 오버로드된다는 점입니다. 예를 들어, 다음은 작동하지 않습니다:

```
> x = 5 + torch.rand(3)
```

덧셈과 뺄셈

한 텐서를 또다른 한 텐서와 + 연산자로 더할 수 있습니다. 뺄셈은 - 연산자로 수행됩니다. 그 텐서들에 있는 요소 개수는 반드시 같아야 합니다. 그러나 차원은 다를 수도 있습니다. 리턴된 텐서의 차원은 첫 텐서의 차원과 같습니다.

```
> x = torch.Tensor(2,2):fill(2)
> y = torch.Tensor(4):fill(3)
> = x+y
 5  5
 5  5
[torch.Tensor of dimension 2x2]

> = y-x
 1
 1
 1
```

```
 1
[torch.Tensor of dimension 4]
```

스칼라 또한 텐서에 더해지거나 빼질 수 있습니다. 그 스칼라는 연산자의 오른쪽에 있어야 합니다.

```
> x = torch.Tensor(2,2):fill(2)
> = x+3
 5  5
 5  5
[torch.Tensor of dimension 2x2]
```

부정 (Negation)

한 텐서는 그 텐서 앞에 - 연산자를 붙임으로써 부정될 수 있습니다.

```
> x = torch.Tensor(2,2):fill(2)
> = -x
-2 -2
-2 -2
[torch.Tensor of dimension 2x2]
```

곱셈

두 텐서 사이 곱은 * 연산자로 지원됩니다. 그 곱셈의 결과는 텐서들의 차원에 따라 달라집니다.

- 1차원과 1차원: 두 텐서의 내적을 리턴합니다 (스칼라).
- 2차원과 1차원: 두 텐서의 행렬-벡터 연산을 리턴합니다 (1차원 텐서).
- 2차원과 2차원: 두 텐서의 행렬-행렬 연산을 리턴합니다 (2차원 텐서).

차원은 반드시 행렬 곱 연산에 맞게 설정되어 있어야 합니다.

또한, 텐서는 스칼라로 곱해질 수도 있습니다. 그 스칼라는 연산자의 왼쪽 또는 오른쪽에 있을 수 있습니다.

예:

```
> M = torch.Tensor(2,2):fill(2)
> N = torch.Tensor(2,4):fill(3)
> x = torch.Tensor(2):fill(4)
> y = torch.Tensor(2):fill(5)
> = x*y -- 내적
40

> = M*x --- 행렬-벡터
 16
 16
[torch.Tensor of dimension 2]

> = M*N -- 행렬-행렬
 12  12  12  12
 12  12  12  12
[torch.Tensor of dimension 2x4]
```

나눗셈

연산자 /는 오직 텐서를 스칼라로 나누는 것만 지원합니다.

예:

```
> x = torch.Tensor(2,2):fill(2)
> = x/3
 0.6667  0.6667
 0.6667  0.6667
[torch.Tensor of dimension 2x2]
```

열 또는 행 방향으로 연산 (차원 방향으로 연산)

[res] torch.cross([res,] a, b [,n])

`y=torch.cross(a,b)`는 길이가 3인 첫 번째 차원을 따라 `a`와 `b`의 벡터곱(cross product)을 리턴합니다.

`y=torch.cross(a,b,n)`는 `n`차원에서 `a`와 `b`의 벡터곱을 리턴합니다.

`a`와 `b`는 반드시 사이즈가 같아야 합니다. 그리고 `a:size(n)`와 `b:size(n)`은 반드시 3이어야 합니다.

[res] torch.cumprod([res,] x [,dim])

`y=torch.cumprod(x)`는 `x`에 있는 요소들의 누적 곱을 리턴합니다. 그 연산은 마지막 차원에서 요소들이 놓여있는 방향으로 수행됩니다.

`y=torch.cumprod(x,n)`는 `x`에 있는 요소들의 누적 곱을 리턴합니다. 그 연산은 차원 `n`에서 요소들이 놓여있는 방향으로 수행됩니다.

```
-- 1. 벡터의 누적 곱
> A=torch.range(1,5)
> A
 1
 2
 3
 4
 5
[torch.DoubleTensor of size 5]
> B=torch.cumprod(A)
> B
   1     -- B(1) = A(1) = 1
   2     -- B(2) = A(1)*A(2) = 1*2 = 2
   6     -- B(3) = A(1)*A(2)*A(3) = 1*2*3 = 6
  24     -- B(4) = A(1)*A(2)*A(3)*A(4) = 1*2*3*4 = 24
 120     -- B(5) = A(1)*A(2)*A(3)*A(4)*A(5) =1*2*3*4*5 = 120
[torch.DoubleTensor of size 5]

-- 2. 행렬의 누적 곱
> A=torch.LongTensor{{1,4,7},{2,5,8},{3,6,9}}
> A
 1  4  7
 2  5  8
 3  6  9
[torch.LongTensor of size 3x3]
> B=torch.cumprod(A)
> B
   1    4    7
   2   20   56
   6  120  504
[torch.LongTensor of size 3x3]
-- Why?
-- B(1,1) = A(1,1) = 1
-- B(2,1) = A(1,1)*A(2,1) = 1*2 = 2
-- B(3,1) = A(1,1)*A(2,1)*A(3,1) = 1*2*3 = 6
-- B(1,2) = A(1,2) = 4
-- B(2,2) = A(1,2)*A(2,2) = 4*5 = 20
-- B(3,2) = A(1,2)*A(2,2)*A(3,2) = 4*5*6 = 120
-- B(1,3) = A(1,3) = 7
-- B(2,3) = A(1,3)*A(2,3) = 7*8 = 56
-- B(3,3) = A(1,3)*A(2,3)*A(3,3) = 7*8*9 = 504
```

```
-- 3. 행렬의 2차원에 대한 누적 곱
> B=torch.cumprod(A,2)
> B
   1    4   28
   2   10   80
   3   18  162
[torch.LongTensor of size 3x3]
-- Why?
-- B(1,1) = A(1,1) = 1
-- B(1,2) = A(1,1)*A(1,2) = 1*4 = 4
-- B(1,3) = A(1,1)*A(1,2)*A(1,3) = 1*4*7 = 28
-- B(2,1) = A(2,1) = 2
-- B(2,2) = A(2,1)*A(2,2) = 2*5 = 10
-- B(2,3) = A(2,1)*A(2,2)*A(2,3) = 2*5*8 = 80
-- B(3,1) = A(3,1) = 3
-- B(3,2) = A(3,1)*A(2,3) = 3*6 = 18
-- B(3,3) = A(3,1)*A(2,3)*A(3,3) = 3*6*9 = 162
```

[res] torch.cumsum([res,] x [,dim])

`y=torch.cumsum(x)`는 `x`에 있는 요소들의 누적 합을 리턴합니다. 그 연산은 첫 번째 차원에서 요소들이 놓여있는 방향으로 수행됩니다.

`y=torch.cumsum(x,n)`는 `x`에 있는 요소들의 누적 곱을 리턴합니다. 그 연산은 차원 `n`에서 요소들이 놓여있는 방향으로 수행됩니다.

torch.max([resval, resind,] x [,dim])

`y=torch.max(x)`는 `x`에서 가장 큰 요소 하나를 리턴합니다.

`y,i=torch.max(x,1)`는 `y`와 `i`를 리턴합니다. `y`에는 `x`의 각 열에서 가장 큰 요소 하나가 들어갑니다. `i`에는 `y`에 있는 요소 값들에 상응하는 `x`에서의 인덱스들로 구성된 텐서가 들어갑니다.

`y,i=torch.max(x,2)`는 각 행에 대한 max 연산을 수행합니다.

`y,i=torch.max(x,n)`는 차원 `n`에 걸쳐 max 연산을 수행합니다.

```
> x=torch.randn(3,3)
> x
 1.1994 -0.6290  0.6888
-0.0038 -0.0908 -0.2075
 0.3437 -0.9948  0.1216
```

```
[torch.DoubleTensor of size 3x3]

th> torch.max(x)
1.1993977428735

th> torch.max(x,1)
 1.1994 -0.0908  0.6888
[torch.DoubleTensor of size 1x3]

 1  2  1
[torch.LongTensor of size 1x3]

th> torch.max(x,2)
 1.1994
-0.0038
 0.3437
[torch.DoubleTensor of size 3x1]

 1
 1
 1
[torch.LongTensor of size 3x1]
```

[res] torch.mean([res,] x [,dim])

`y=torch.mean(x)`는 `x`에 있는 모든 요소들의 평균을 리턴합니다.

`y=torch.mean(x,1)`는 `x`에 있는 각 열에 있는 요소들의 평균으로 구성된 텐서 `y`를 리턴합니다.

`y=torch.mean(x,2)`는 각 행을 위한 평균 연산을 수행합니다.

`y=torch.mean(x,n)`는 차원 `n`에 걸친 평균 연산을 수행합니다.

torch.min([resval, resind,] x [,dim])

`y=torch.min(x)`는 `x`에 있는 가장 작은 요소 하나를 리턴합니다.

`y,i=torch.min(x,1)`는 `y`와 `i`를 리턴합니다. `y`는 `x`의 각 열에서 가장 작은 요소 하나입니다. `i`는 `x`의 각 열에서 가장 작은 요소가 있는 인덱스들로 구성된 텐서입니다.

`y,i=torch.min(x,2)`는 각 행에 대한 min 연산을 수행합니다.

y,i=torch.min(x,n)는 min 연산을 차원 n에 걸쳐 수행합니다.

[res] torch.cmax([res,] tensor1, tensor2)

tensor1과 tensor2에 있는 상응하는 각 두 요소별 최댓값들로 구성된 텐서 하나를 리턴합니다.

c=torch.cmax(a, b)는 a와 b의 요소별 최댓값들로 구성된 새 텐서 하나를 리턴합니다.

a:cmax(b)는 a와 b의 요소별 최댓값을 a에 저장합니다.

c:max(a, b)는 a와 b의 요소별 최댓값을 c에 저장합니다.

```
> a = torch.Tensor{1, 2, 3}
> b = torch.Tensor{3, 2, 1}
> = torch.cmax(a, b)
 3
 2
 3
[torch.DoubleTensor of size 3]
```

[res] torch.cmax([res,] tensor, value)

tensor에 있는 각 요솟값과 value 사이의 최댓값을 계산합니다.

c=torch.cmax(a, v)는 v와 a의 각 요솟값 사이의 최댓값들로 구성된 새 텐서 하나를 리턴합니다.

a:cmax(v)는 a의 각 요솟값과 v 사이의 최댓값들을 a에 저장합니다.

c:max(a, v)는 a의 각 요솟값과 v 사이의 최댓값들을 c에 저장합니다.

```
> a = torch.Tensor{1, 2, 3}
> = torch.cmax(a, 2)
 2
 2
 3
[torch.DoubleTensor of size 3]
```

[res] torch.cmin([res,] tensor1, tensor2)

tensor1과 tensor2에 있는 상응하는 각 두 요소의 최솟값들로 구성된 텐서 하나를 리턴합니다.

c=torch.cmin(a, b)는 a와 b의 요소별 최솟값을 포함하는 새 텐서 하나를 리턴합니다.

a:cmin(b)는 a와 b의 요소별 최솟값을 a에 저장합니다.

c:min(a, b)는 a와 b의 요소별 최솟값을 c에 저장합니다.

```
> a = torch.Tensor{1, 2, 3}
> b = torch.Tensor{3, 2, 1}
> = torch.cmin(a, b)
 1
 2
 1
[torch.DoubleTensor of size 3]
```

[res] torch.cmin([res,] tensor, value)

tensor에 있는 각 요솟값과 value 사이의 최솟값을 계산합니다.

c=torch.cmin(a, v)는 v와 a의 각 요솟값 사이의 최솟값들로 구성된 새 텐서 하나를 리턴합니다.

a:cmin(v)는 a의 각 요솟값과 v 사이의 최솟값들을 a에 저장합니다.

c:min(a, v)는 a의 각 요솟값과 v 사이의 최솟값들을 c에 저장합니다.

```
> a = torch.Tensor{1, 2, 3}
> = torch.cmin(a, 2)
 1
 2
 2
[torch.DoubleTensor of size 3]
```

torch.median([resval, resind,] x [,dim])

y=torch.median(x)는 x의 마지막 차원에 대한 중간 값(요소 수가 짝수인 경우 중간보다 하나 앞) 요소를 리턴합니다.

`y,i=torch.median(x,1)`는 `y`와 `i`를 리턴합니다. `y`에는 `x`의 각 열(행들을 건너뛰는)에 있는 중간 값이 리턴됩니다. 한 텐서 `i`에는 `y`에 있는 각 중간 값에 상응하는 `x`의 각 열에서의 인덱스들이 리턴됩니다.

`y,i=torch.median(x,2)`는 중간 값 연산을 각 행별로 수행합니다.

`y,i=torch.median(x,n)`는 중간 값 연산을 차원 `n`에 대해 수행합니다.

```
> x=torch.randn(3,3)
> x
 0.7860  0.7687 -0.9362
 0.0411  0.5407 -0.3616
-0.0129 -0.2499 -0.5786
[torch.DoubleTensor of size 3x3]

> y,i=torch.median(x)
> y
 0.7687
 0.0411
-0.2499
[torch.DoubleTensor of size 3x1]
> i
 2
 1
 2
[torch.LongTensor of size 3x1]

> y,i=torch.median(x,1)
> y
 0.0411  0.5407 -0.5786
[torch.DoubleTensor of size 1x3]
> i
 2  2  3
[torch.LongTensor of size 1x3]

> y,i=torch.median(x,2)
> y
 0.7687
 0.0411
-0.2499
[torch.DoubleTensor of size 3x1]
> i
 2
 1
 2
[torch.LongTensor of size 3x1]
```

torch.mode([resval, resind,] x [,dim])

y=torch.mode(x)는 x의 마지막 차원에 대해 가장 빈번하게 나오는 요소를 리턴합니다.

y,i=torch.mode(x,1)는 y와 i를 리턴합니다. y에는 x에 있는 각 열의 모드 요소가 리턴됩니다. i에는 y에 있는 각 값에 상응하는 x의 각 열에서의 인덱스들이 리턴됩니다.

y,i=torch.mode(x,2)는 모드 연산을 각 행에 대해 수행합니다.

y,i=torch.mode(x,n)는 모드 연산을 차원 n에 대해 수행합니다.

torch.kthvalue([resval, resind,] x, k [,dim])

y=torch.kthvalue(x,k)는 x의 마지막 차원에 대한 k 번째로 작은 요소를 리턴합니다.

y,i=torch.kthvalue(x,k,1)는 y와 i를 리턴합니다. y에는 x의 각 열에서 k 번째로 작은 요소가 리턴됩니다. i에는 y에 있는 각 값에 상응하는 x의 각 열에서의 인덱스들이 리턴됩니다.

y,i=torch.kthvalue(x,k,2)는 k 번째 값 연산을 각 행에 대해 수행합니다.

y,i=torch.kthvalue(x,k,n)는 k 번째 값 연산을 차원 n에 대해 수행합니다.

[res] torch.prod([res,] x [,n])

y=torch.prod(x)는 x에 있는 모든 요소의 곱을 리턴합니다.

y=torch.prod(x,n)는 한 텐서 y를 리턴합니다. 차원 n의 크기는 1로 줄어듭니다. 그리고 그 줄어든 요소가 있는 자리에 줄어든 차원 n에 걸쳐있던 x의 요소들의 곱이 저장됩니다.

```
> a = torch.Tensor{{{1,2},{3,4}}, {{5,6},{7,8}}}
> a
(1,.,.) =
  1  2
  3  4

(2,.,.) =
  5  6
  7  8
[torch.DoubleTensor of dimension 2x2x2]

> torch.prod(a, 1)
```

```
(1,.,.) =
   5  12
  21  32
[torch.DoubleTensor of dimension 1x2x2]

> torch.prod(a, 2)
(1,.,.) =
   3   8
(2,.,.) =
  35  48
[torch.DoubleTensor of size 2x1x2]

> torch.prod(a, 3)
(1,.,.) =
   2
  12
(2,.,.) =
  30
  56
[torch.DoubleTensor of size 2x2x1]
```

torch.sort([resval, resind,] x [,d] [,flag])

`y,i=torch.sort(x)`는 `y`와 `i`를 리턴합니다. `y`에는 `x`의 모든 항목들이 마지막 차원을 따라 **오름차순으로** 정렬된 결과가 리턴됩니다. 한 텐서 `i`에는 `y`에 있는 요소들에 상응하는 `x`에서의 인덱스들이 리턴됩니다.

`y,i=torch.sort(x,d)`는 특정 차원 `d`를 따라 정렬 연산을 수행합니다.

`y,i=torch.sort(x)`는 그러므로 `y,i=torch.sort(x,x:dim())`와 같습니다.

`y,i=torch.sort(x,d,true)`는 특정 차원 `d`를 따라 **내림차순으로** 정렬 연산을 수행합니다.

```
> x=torch.randn(3,3)
> x
-1.2470 -0.4288 -0.5337
 0.8836 -0.1622  0.9604
 0.6297  0.2397  0.0746
[torch.DoubleTensor of size 3x3]

> torch.sort(x)
-1.2470 -0.5337 -0.4288
-0.1622  0.8836  0.9604
 0.0746  0.2397  0.6297
[torch.DoubleTensor of size 3x3]

 1  3  2
 2  1  3
```

```
 3  2  1
[torch.LongTensor of size 3x3]
```

[res] torch.std([res,] x, [,dim] [,flag])

`y=torch.std(x)`는 `x`에 있는 요소들의 표준 편차를 리턴합니다.

`y=torch.std(x,dim)`는 차원 `dim`에 대해 `std` 연산을 수행합니다.

`y=torch.std(x,dim,false)`는 `n-1`으로 정규화된 `std` 연산을 수행합니다 (이것이 기본값입니다).

`y=torch.std(x,dim,true)`는 `n-1` 대신 `n`으로 정규화된 `std` 연산을 수행합니다.

[res] torch.sum([res,] x)

`y=torch.sum(x)`는 `x`에 있는 요소들의 합을 리턴합니다.

`y=torch.sum(x,2)`는 `x`의 각 열에 대한 요소들의 합을 리턴합니다.

`y=torch.sum(x,n)`는 `x`의 차원 `n`에 대한 요소들의 합을 리턴합니다.

[res] torch.var([res,] x [,dim] [,flag])

`y=torch.var(x)`는 `x`에 있는 요소들의 분산을 리턴합니다.

`y=torch.var(x,dim)`는 차원 `dim`에 걸쳐 `var` 연산을 수행합니다.

`y=torch.var(x,dim,false)`는 `n-1`으로 정규화된 `var` 연산을 수행합니다 (이것이 기본값입니다).

`y=torch.var(x,dim,true)`는 `n-1`대신 `n`으로 정규화된 `var` 연산을 수행합니다.

행렬 전체에 대한 연산 (텐서 전체에 대한 연산); Matrix-wide operations (tensor-wide operations)

유념하십시오. 차원별 연산들에 있는 연산들도 단지 `dim` 파라미터를 생략함으로써 행렬 전체에 대한 연산들에 사용될 수 있습니다.

torch.norm(x [,p] [,dim])

`y=torch.norm(x)`는 `x`의 2놈(norm)을 리턴합니다.

`y=torch.norm(x,p)`는 텐서 `x`의 `p`놈을 리턴합니다.

`y=torch.norm(x,p,dim)`는 차원 `dim`에 걸쳐 계산된 텐서 `x`의 `p`놈을 리턴합니다.

torch.renorm([res], x, p, dim, maxnorm)

차원 `dim`을 따라 `maxnorm` 놈을 초과하는 서브텐서들을 renormalize합니다.

`y=torch.renorm(x,p,dim,maxnorm)`는 `dim` 차원을 제외한 모든 차원들에 걸쳐 `maxnorm`을 넘지 않는 `p`놈을 가진 `x`의 한 버전을 리턴합니다. `dim` 인자를 함수 norm에 있는 같은 이름을 가진 인자와 헷갈리지 않아야 합니다. 이 경우, 각 `i` 번째 서브텐서 `x:select(dim,i)`를 위해 `p`놈이 측정됩니다. 이 함수는 다음과 같습니다 (그러나 더 빠릅니다).

```
function renorm(matrix, value, dim, maxnorm)
  local m1 = matrix:transpose(dim, 1):contiguous()
  -- collapse non-dim dimensions:
  m2 = m1:reshape(m1:size(1), m1:nElement()/m1:size(1))
  local norms = m2:norm(value,2)
  -- clip
  local new_norms = norms:clone()
  new_norms[torch.gt(norms, maxnorm)] = maxnorm
  new_norms:cdiv(norms:add(1e-7))
  -- renormalize
  m1:cmul(new_norms:expandAs(m1))
  return m1:transpose(dim, 1)
end
```

`x:renorm(p,dim,maxnorm)`는 `x:copy(torch.renorm(x,p,dim,maxnorm))`와 같은 결과를 리턴합니다.

주의: 이 함수는 파라미터 텐서들의 놈을 제한하기 위한 레귤러라이저(regularizer)로서 특히 유용합니다. 힌튼 외 2012, 2 쪽을 보십시오.

torch.dist(x,y)

`y=torch.dist(x,y)`는 `x-y`의 2놈을 리턴합니다.

`y=torch.dist(x,y,p)`는 `x-y`의 `p`놈을 리턴합니다.

torch.numel(x)

`y=torch.numel(x)`는 행렬 `x`에 있는 요소들의 개수를 리턴합니다.

torch.trace(x)

`y=torch.trace(x)`는 한 행렬 `x`의 대각합(대각 요소들의 합)을 리턴합니다. 이것은 `x`의 고윳값들의 합과 같습니다. 리턴되는 값 `y`는 (텐서 하나가 아니라) 숫자 하나입니다.

컨볼루션 연산

이 함수들은 한 커널(또는 커널들의 집합)을 가지고 한 입력 영상(또는 입력 영상들의 집합)의 컨볼루션 또는 상호 상관을 구현합니다. 토치의 컨볼루션 함수는 타입이 다른 입력/커널 차원들을 다룰 수 있습니다. 그리고 상응하는 출력들을 생성할 수 있습니다. 연산들의 일반적 형태는 항상 같습니다.

[res] torch.conv2([res,] x, k, [, 'F' or 'V'])

이 함수는 `x`와 `k` 사이의 2차원 컨볼루션을 계산합니다. 입력과 커널의 차원 수가 2로 줄어들 때, 이 연산들은 BLAS 연산과 비슷합니다.

- `x`와 `k`가 2차원 : 한 커널을 가진 한 영상의 컨볼루션 (2차원 출력). 이 연산은 두 스칼라의 곱과 비슷합니다.
- `x(p x m x n)`와 `k(p x ki x kj)`가 3차원 : 상응하는 커널을 가진 각 입력 슬라이스의 컨볼루션 (3차원 출력).

- x(p x m x n) 3차원, k(q x p x ki x kj) 4차원 : 상응하는 커널의 슬라이스를 가진 모든 입력 슬라이스들의 컨볼루션. 출력은 3차원(q x m x n). 이 연산은 행렬 k와 벡터 x의 행렬 벡터 곱과 비슷합니다.

마지막 인자는 그 컨볼루션이 full('F') 컨볼루션인지 valid('V') 컨볼루션인지를 제어합니다. 기본값은 **valid** 컨볼루션입니다.

```
x = torch.rand(100,100)
k = torch.rand(10,10)
c = torch.conv2(x,k)
> c:size()
 91
 91
[torch.LongStorage of size 2]

c = torch.conv2(x,k,'F')
> c:size()
 109
 109
[torch.LongStorage of size 2]
```

[res] torch.xcorr2([res,] x, k, [, 'F' or 'V'])

이 함수는 torch.conv2와 같은 옵션들과 입력/출력 구성들을 가집니다. 그러나 입력과 커널 k의 상호 상관을 계산합니다.

[res] torch.conv3([res,] x, k, [, 'F' or 'V'])

이 함수는 x와 k 사이의 3차원 컨볼루션을 계산합니다. 입력과 커널의 차원들의 수가 3으로 줄어들 때, 이 연산들은 BLAS 연산들과 비슷합니다.

- x와 k가 3차원 : 한 커널을 가진 한 영상의 컨볼루션 (3차원 출력). 이 연산은 두 스칼라의 곱과 비슷합니다.
- x(p x m x n x o)와 k(p x ki x kj x kk)가 4차원 : 상응하는 커널을 가진 각 입력 슬라이스의 컨볼루션 (4차원 출력).

- x(p x m x n x o) 4차원, k(q x p x ki x kj x kk) 5차원 : 상응하는 커널의 슬라이스를 가진 모든 입력 슬라이스들의 컨볼루션. 출력은 4차원(q x m x n x o). 이 연산은 행렬 k와 벡터 x의 행렬 벡터곱과 비슷합니다.

마지막 인자는 그 컨볼루션이 full('F') 컨볼루션인지 valid('V') 컨볼루션인지를 제어합니다. 기본값은 **valid** 컨볼루션입니다.

```
x = torch.rand(100,100,100)
k = torch.rand(10,10,10)
c = torch.conv3(x,k)
> c:size()
 91
 91
 91
[torch.LongStorage of size 3]

c = torch.conv3(x,k,'F')
> c:size()
 109
 109
 109
[torch.LongStorage of size 3]
```

[res] torch.xcorr3([res,] x, k, [, 'F' or 'V'])

이 함수는 torch.conv3와 같은 옵션들과 입력/출력 구성들을 가집니다. 그러나 입력과 커널 k의 상호상관을 수행합니다.

고윳값, SVD, 선형 시스템 솔루션

이 절의 함수들은 라팩(LAPACK) 라이브러리 인터페이스로 구현됩니다. 만약 컴파일 단계에서 라팩 라이브러리들이 없으면, 이 함수들을 쓸 수 없습니다.

[x,lu] torch.gesv([resb, resa,] B, A)

X,LU=torch.gesv(B,A)는 AX=B의 정답과 LU을 리턴합니다. LU는 A의 LU 분해를 위한 L과 U 인자를 담고있습니다.

A는 반드시 정방 행렬 이고 비특이 행렬(2차원 텐서)이어야 합니다. A와 LU는 m x m이고, X는 m x k이고, B는 m x k입니다.

만약 resb와 resa가 주어지면, resb와 resa는 임시 스토리지와 결과 리턴을 위해 사용됩니다.

- resa는 A의 LU 분해를 위한 L과 U 인자를 담습니다.
- resb는 정답 X를 담습니다.

주의: 원래 strides를 개의치 않고, 리턴된 행렬 resb와 resa는 전치(transpose)됩니다. 다시 말해, strides로 m,1 대신 1,m을 가집니다.

```
> a = torch.Tensor({{6.80, -2.11,  5.66,  5.97,  8.23},
                    {-6.05, -3.30,  5.36, -4.44,  1.08},
                    {-0.45,  2.58, -2.70,  0.27,  9.04},
                    {8.32,  2.71,  4.35,  -7.17,  2.14},
                    {-9.67, -5.14, -7.26,  6.08, -6.87}}):t()

> b = torch.Tensor({{4.02,  6.19, -8.22, -7.57, -3.03},
                    {-1.56,  4.00, -8.67,  1.75,  2.86},
                    {9.81, -4.09, -4.57, -8.61,  8.99}}):t()

> b
 4.0200 -1.5600  9.8100
 6.1900  4.0000 -4.0900
-8.2200 -8.6700 -4.5700
-7.5700  1.7500 -8.6100
-3.0300  2.8600  8.9900
[torch.DoubleTensor of dimension 5x3]

> a
 6.8000 -6.0500 -0.4500  8.3200 -9.6700
-2.1100 -3.3000  2.5800  2.7100 -5.1400
 5.6600  5.3600 -2.7000  4.3500 -7.2600
 5.9700 -4.4400  0.2700 -7.1700  6.0800
 8.2300  1.0800  9.0400  2.1400 -6.8700
[torch.DoubleTensor of dimension 5x5]


> x = torch.gesv(b,a)
> x
-0.8007 -0.3896  0.9555
-0.6952 -0.5544  0.2207
 0.5939  0.8422  1.9006
 1.3217 -0.1038  5.3577
 0.5658  0.1057  4.0406
[torch.DoubleTensor of dimension 5x3]
```

```
> b:dist(a*x)
1.1682163181673e-14
```

[x] torch.trtrs([resb, resa,] b, a [, 'U' or 'L'] [, 'N' or 'T'] [, 'N' or 'U'])

`X=torch.trtrs(B,A)`는 `AX=B`의 정답을 리턴합니다. 여기서 `A`는 상삼각(upper-triangular) 입니다.

`A`는 반드시 정방, 삼각, 비특히 행렬(2차원 텐서)이어야 합니다. `A`와 `resa`는 `m x m`이고, `X`와 `B`는 `m x k`입니다. (정확히 말하면, `A`가 반드시 삼각이고 비특이일 필요는 없습니다. 오직 `A`의 상 또는 하 삼각이 고려될 것이므로 그 부분만 비특이이면 됩니다.)

이 함수는 몇 가지 옵션들을 가집니다.

- `uplo('U' 또는 'L')`는 `A`가 상삼각인지 하삼각인지를 특정합니다. 기본값은 `'U'`입니다.
- `trans('N' 또는 'T')`는 그 시스템 또는 등식을 `'N'` 또는 `'T'`로 특정합니다. `'N'`은 `A * X = B`(전치 없음)를, `'T'`는 `A^T * X = B`(전치)를 나타냅니다. 기본값은 `'N'`입니다.
- `diag('N' 또는 'U')`는 `A`가 단위 삼각인지 아닌지를 특정합니다. `'U'`는 `A`가 단위 삼각임을 나타냅니다. 다시 말해, `A`는 대각 요소들로 1을 가집니다. `'N'`은 `A`가 꼭 단위 삼각일 필요는 없음을 나타냅니다. 기본값은 `'N'`입니다.

만약 `resb`와 `resa`가 주어지면, `resb`와 `resa`는 임시 스토리지와 결과를 리턴하기 위해 사용됩니다. `resb`는 `X`의 정답을 담습니다.

주의: 원래 strides를 개의치 않고, 리턴된 행렬 `resb`와 `resa`는 전치됩니다. 다시 말해, strides로 `m,1` 대신 `1,m`을 가집니다.

```
> a = torch.Tensor({{6.80, -2.11,  5.66,  5.97,  8.23},
                    {0, -3.30,  5.36, -4.44,  1.08},
                    {0,  0, -2.70,  0.27,  9.04},
                    {0,  0,  0,  -7.17,  2.14},
                    {0,  0,  0,  0, -6.87}})

> b = torch.Tensor({{4.02,  6.19, -8.22, -7.57, -3.03},
                    {-1.56,  4.00, -8.67,  1.75,  2.86},
                    {9.81, -4.09, -4.57, -8.61,  8.99}}):t()

> b
```

```
 4.0200 -1.5600  9.8100
 6.1900  4.0000 -4.0900
-8.2200 -8.6700 -4.5700
-7.5700  1.7500 -8.6100
-3.0300  2.8600  8.9900
[torch.DoubleTensor of dimension 5x3]

> a
 6.8000 -2.1100  5.6600  5.9700  8.2300
 0.0000 -3.3000  5.3600 -4.4400  1.0800
 0.0000  0.0000 -2.7000  0.2700  9.0400
 0.0000  0.0000  0.0000 -7.1700  2.1400
 0.0000  0.0000  0.0000  0.0000 -6.8700
[torch.DoubleTensor of dimension 5x5]


> x = torch.trtrs(b, a)
> x
-3.5416 -0.2514  3.0847
 4.2072  2.0391 -4.5146
 4.6399  1.7804 -2.6077
 1.1874 -0.3683  0.8103
 0.4410 -0.4163 -1.3086
[torch.DoubleTensor of size 5x3]

> b:dist(a*x)
4.1895292266754e-15
```

torch.potrf([res,] A [, 'U' or 'L'])

2차원 텐서 `A`의 촐레스키 분해. 행렬 A는 반드시 정부호(positive-definite)이고 대칭(symetric) 또는 복소 에르미트(Hermitian)이어야 합니다.

선택적 문자 `uplo` = {'U', 'L'}는 리턴될 행렬이 상삼각이어야 하는지 하삼각이어야 하는지를 특정합니다. 기본값은 `uplo` = 'U'입니다.

`X = torch.potrf(A, 'U')`는 X의 상삼각 촐레스키 분해를 리턴합니다.

`X = torch.potrf(A, 'L')`는 X의 하삼각 촐레스키 분해를 리턴합니다.

만약 텐서 `res`가 주어지면, 분해 결과가 `res`에 저장됩니다.

```
> A = torch.Tensor({
    {1.2705,  0.9971,  0.4948,  0.1389,  0.2381},
    {0.9971,  0.9966,  0.6752,  0.0686,  0.1196},
```

```
    {0.4948,  0.6752,  1.1434,  0.0314,  0.0582},
    {0.1389,  0.0686,  0.0314,  0.0270,  0.0526},
    {0.2381,  0.1196,  0.0582,  0.0526,  0.3957}})

> chol = torch.potrf(A)
> chol
 1.1272  0.8846  0.4390  0.1232  0.2112
 0.0000  0.4626  0.6200 -0.0874 -0.1453
 0.0000  0.0000  0.7525  0.0419  0.0738
 0.0000  0.0000  0.0000  0.0491  0.2199
 0.0000  0.0000  0.0000  0.0000  0.5255
[torch.DoubleTensor of size 5x5]

> torch.potrf(chol, A, 'L')
> chol
 1.1272  0.0000  0.0000  0.0000  0.0000
 0.8846  0.4626  0.0000  0.0000  0.0000
 0.4390  0.6200  0.7525  0.0000  0.0000
 0.1232 -0.0874  0.0419  0.0491  0.0000
 0.2112 -0.1453  0.0738  0.2199  0.5255
[torch.DoubleTensor of size 5x5]
```

torch.potrs([res,] B, chol [, 'U' or 'L'])

2차원 텐서 `A`의 촐레스키 분해 `chol`을 사용하여 선형 시스템 `AX = B`의 정답을 리턴합니다.

정방 행렬 `chol`은 꼭 삼각이어야 합니다. 그리고 오른쪽 항에 있는 행렬 `B`는 꼭 full rank여야 합니다.

선택적 문자 `uplo` = {'U', 'L'}는 행렬 `chol`이 상삼각인지 하삼각인지를 특정합니다. 기본값은 'U'입니다.

만약 텐서 `res`가 주어지면, 분해 결과가 `res`에 저장됩니다.

```
> A = torch.Tensor({
    {1.2705,  0.9971,  0.4948,  0.1389,  0.2381},
    {0.9971,  0.9966,  0.6752,  0.0686,  0.1196},
    {0.4948,  0.6752,  1.1434,  0.0314,  0.0582},
    {0.1389,  0.0686,  0.0314,  0.0270,  0.0526},
    {0.2381,  0.1196,  0.0582,  0.0526,  0.3957}})

> B = torch.Tensor({
    {0.6219,  0.3439,  0.0431},
    {0.5642,  0.1756,  0.0153},
    {0.2334,  0.8594,  0.4103},
    {0.7556,  0.1966,  0.9637},
    {0.1420,  0.7185,  0.7476}})

> chol = torch.potrf(A)
> chol
```

```
 1.1272  0.8846  0.4390  0.1232  0.2112
 0.0000  0.4626  0.6200 -0.0874 -0.1453
 0.0000  0.0000  0.7525  0.0419  0.0738
 0.0000  0.0000  0.0000  0.0491  0.2199
 0.0000  0.0000  0.0000  0.0000  0.5255
[torch.DoubleTensor of size 5x5]

> solve = torch.potrs(B, chol)
> solve
  12.1945   61.8622   92.6882
 -11.1782  -97.0303 -138.4874
 -15.3442  -76.6562 -116.8218
   6.1930   13.5238   25.2056
  29.9678  251.7346  360.2301
[torch.DoubleTensor of size 5x3]

> A*solve
 0.6219  0.3439  0.0431
 0.5642  0.1756  0.0153
 0.2334  0.8594  0.4103
 0.7556  0.1966  0.9637
 0.1420  0.7185  0.7476
[torch.DoubleTensor of size 5x3]

> B:dist(A*solve)
4.6783066076306e-14
```

torch.potri([res,] chol [, 'U' or 'L'])

촐레스키 분해 `chol`이 주어진 2차원 텐서 `A`의 역행렬(inverse)을 리턴합니다.

정방 행렬 `chol`은 삼각 행렬이어야 합니다.

선택적 문자 `uplo` = {'U', 'L'}는 행렬 `chol`이 상삼각인지 하삼각인지를 특정합니다. 기본값은 'U'입니다.

만약 텐서 `res`가 주어지면, 결과 역행렬이 `res`에 저장됩니다.

```
> A = torch.Tensor({
    {1.2705,  0.9971,  0.4948,  0.1389,  0.2381},
    {0.9971,  0.9966,  0.6752,  0.0686,  0.1196},
    {0.4948,  0.6752,  1.1434,  0.0314,  0.0582},
    {0.1389,  0.0686,  0.0314,  0.0270,  0.0526},
    {0.2381,  0.1196,  0.0582,  0.0526,  0.3957}})

> chol = torch.potrf(A)
> chol
 1.1272  0.8846  0.4390  0.1232  0.2112
 0.0000  0.4626  0.6200 -0.0874 -0.1453
```

```
 0.0000  0.0000  0.7525  0.0419  0.0738
 0.0000  0.0000  0.0000  0.0491  0.2199
 0.0000  0.0000  0.0000  0.0000  0.5255
[torch.DoubleTensor of size 5x5]

> inv = torch.potri(chol)
> inv
  42.2781  -39.0824    8.3019 -133.4998    2.8980
 -39.0824   38.1222   -8.7468  119.4247   -2.5944
   8.3019   -8.7468    3.1104  -25.1405    0.5327
-133.4998  119.4247  -25.1405  480.7511  -15.9747
   2.8980   -2.5944    0.5327  -15.9747    3.6127
[torch.DoubleTensor of size 5x5]

> inv:dist(torch.inverse(A))
2.8525852877633e-12
```

torch.gels([resb, resa,] b, a)

Full rank `m x n` 행렬 `A`를 위한 최소 제곱법과 least norm problem의 해.

- 만약 `n <= m`이면, `||AX-B||_F`을 풉니다.
- 만약 `n > m`이면, `min ||X||_F` s.t. `AX=B`를 풉니다.

리턴에서, 행렬 `x`의 첫 `n` 행들은 해를 포함합니다, 그리고 나머지는 그 나머지 정보를 포함합니다. 행 `n + 1`에서 시작하는 `x`의 각 열의 요소들의 제곱의 합의 제곱근은 상응하는 열을 위한 residual입니다.

주의: 원래 strides를 개의치 않고, 리턴된 행렬 `resb`와 `resa`는 전치됩니다. 다시 말해, strides로 `m,1` 대신 `1,m`을 가집니다.

```
> a = torch.Tensor({{ 1.44, -9.96, -7.55,  8.34,  7.08, -5.45},
                    {-7.84, -0.28,  3.24,  8.09,  2.52, -5.70},
                    {-4.39, -3.24,  6.27,  5.28,  0.74, -1.19},
                    {4.53,  3.83, -6.64,  2.06, -2.47,  4.70}}):t()

> b = torch.Tensor({{8.58,  8.26,  8.48, -5.28,  5.72,  8.93},
                    {9.35, -4.43, -0.70, -0.26, -7.36, -2.52}}):t()

> a
 1.4400 -7.8400 -4.3900  4.5300
-9.9600 -0.2800 -3.2400  3.8300
-7.5500  3.2400  6.2700 -6.6400
 8.3400  8.0900  5.2800  2.0600
 7.0800  2.5200  0.7400 -2.4700
-5.4500 -5.7000 -1.1900  4.7000
[torch.DoubleTensor of dimension 6x4]
```

```
> b
 8.5800  9.3500
 8.2600 -4.4300
 8.4800 -0.7000
-5.2800 -0.2600
 5.7200 -7.3600
 8.9300 -2.5200
[torch.DoubleTensor of dimension 6x2]

> x = torch.gels(b,a)
> x
 -0.4506   0.2497
 -0.8492  -0.9020
  0.7066   0.6323
  0.1289   0.1351
 13.1193  -7.4922
 -4.8214  -7.1361
[torch.DoubleTensor of dimension 6x2]

> b:dist(a*x:narrow(1,1,4))
17.390200628863

> math.sqrt(x:narrow(1,5,2):pow(2):sumall())
17.390200628863
```

torch.symeig([rese, resv,] a [, 'N' or 'V'] [, 'U' or 'L'])

`e,V=torch.symeig(A)`는 한 실수 행렬 `A`의 고윳값들과 고유벡터들를 리턴합니다.

`A`와 `V`는 `m x m` 행렬입니다. `e`는 `m` 차원 벡터 하나입니다.

이 함수는 `A = V' diag(e) V`인 `A`의 모든 고윳값들(과 고유벡터들)을 계산합니다.

세 번째 인자는 고윳값들만 계산할지 고유벡터들도 함께 계산할지를 정의합니다. 만약 세 번째 인자가 `'N'`이면, 오직 고윳값들만 계산됩니다. 만약 세 번째 인자가 `'V'`이면, 고윳값들과 고유벡터들이 모두 계산됩니다.

입력 행렬 `A`가 대칭이라고 가정되었으므로, 오직 상삼각 부분이 사용됩니다 (기본값). 만약 네 번재 인자가 `'L'`이면, 하삼각 부분이 사용됩니다.

주의: 원래 strides를 개의치 않고, 리턴되는 행렬 `v`는 전치됩니다. 다시 말해, `m,1` 대신 `1,m` strides를 가집니다.

```
> a = torch.Tensor({{ 1.96,  0.00,  0.00,  0.00,  0.00},
                    {-6.49,  3.80,  0.00,  0.00,  0.00},
                    {-0.47, -6.39,  4.17,  0.00,  0.00},
                    {-7.20,  1.50, -1.51,  5.70,  0.00},
                    {-0.65, -6.34,  2.67,  1.80, -7.10}}):t()

> a
 1.9600 -6.4900 -0.4700 -7.2000 -0.6500
 0.0000  3.8000 -6.3900  1.5000 -6.3400
 0.0000  0.0000  4.1700 -1.5100  2.6700
 0.0000  0.0000  0.0000  5.7000  1.8000
 0.0000  0.0000  0.0000  0.0000 -7.1000
[torch.DoubleTensor of dimension 5x5]

> e = torch.symeig(a)
> e
-11.0656
 -6.2287
  0.8640
  8.8655
 16.0948
[torch.DoubleTensor of dimension 5]

> e,v = torch.symeig(a,'V')
> e
-11.0656
 -6.2287
  0.8640
  8.8655
 16.0948
[torch.DoubleTensor of dimension 5]

> v
-0.2981 -0.6075  0.4026 -0.3745  0.4896
-0.5078 -0.2880 -0.4066 -0.3572 -0.6053
-0.0816 -0.3843 -0.6600  0.5008  0.3991
-0.0036 -0.4467  0.4553  0.6204 -0.4564
-0.8041  0.4480  0.1725  0.3108  0.1622
[torch.DoubleTensor of dimension 5x5]

> v*torch.diag(e)*v:t()
 1.9600 -6.4900 -0.4700 -7.2000 -0.6500
-6.4900  3.8000 -6.3900  1.5000 -6.3400
-0.4700 -6.3900  4.1700 -1.5100  2.6700
-7.2000  1.5000 -1.5100  5.7000  1.8000
-0.6500 -6.3400  2.6700  1.8000 -7.1000
[torch.DoubleTensor of dimension 5x5]
```

```
> a:dist(torch.triu(v*torch.diag(e)*v:t()))
1.0219480822443e-14
```

torch.eig([rese, resv,] a [, 'N' or 'V'])

`e,V=torch.eig(A)`는 한 일반 실수 정방 행렬 `A`의 고윳값과 고유벡터를 리턴합니다.

`A`와 `V`는 `m x m` 행렬이고 `e` 는 `m` 차원 벡터입니다.

이 함수는 `A = V' diag(e) V`인 의 `A`의 모든 우 고윳값들 (그리고 고유벡터들)을 계산합니다.

세 번째 인자는 고윳값의 고유벡터까지 계산할지를 특정합니다. 만약 그 인자가 `'N'`이면, 오직 고윳값들만 계산됩니다. 만약 그 인자가 `'V'`이면, 고윳값과 고유벡터가 모두 계산됩니다.

그 고윳값들은 LAPACK 컨벤션에 따르고 복소수 (실수/허수) 쌍으로 된 수들로 리턴됩니다 (`2*m` 차원 텐서).

주의: 원래 strides를 개의치 않고, 리턴된 행렬 `V`는 전치됩니다. 다시 말해, strides로 `m,1` 대신 `1,m`을 가집니다.

```
> a = torch.Tensor({{ 1.96,  0.00,  0.00,  0.00,  0.00},
                    {-6.49,  3.80,  0.00,  0.00,  0.00},
                    {-0.47, -6.39,  4.17,  0.00,  0.00},
                    {-7.20,  1.50, -1.51,  5.70,  0.00},
                    {-0.65, -6.34,  2.67,  1.80, -7.10}}):t()

> a
 1.9600 -6.4900 -0.4700 -7.2000 -0.6500
 0.0000  3.8000 -6.3900  1.5000 -6.3400
 0.0000  0.0000  4.1700 -1.5100  2.6700
 0.0000  0.0000  0.0000  5.7000  1.8000
 0.0000  0.0000  0.0000  0.0000 -7.1000
[torch.DoubleTensor of dimension 5x5]

> b = a + torch.triu(a,1):t()
> b

  1.9600 -6.4900 -0.4700 -7.2000 -0.6500
 -6.4900  3.8000 -6.3900  1.5000 -6.3400
 -0.4700 -6.3900  4.1700 -1.5100  2.6700
 -7.2000  1.5000 -1.5100  5.7000  1.8000
 -0.6500 -6.3400  2.6700  1.8000 -7.1000
[torch.DoubleTensor of dimension 5x5]
```

```
> e = torch.eig(b)
> e
 16.0948   0.0000
-11.0656   0.0000
 -6.2287   0.0000
  0.8640   0.0000
  8.8655   0.0000
[torch.DoubleTensor of dimension 5x2]

> e,v = torch.eig(b,'V')
> e
 16.0948   0.0000
-11.0656   0.0000
 -6.2287   0.0000
  0.8640   0.0000
  8.8655   0.0000
[torch.DoubleTensor of dimension 5x2]

> v
-0.4896  0.2981 -0.6075 -0.4026 -0.3745
 0.6053  0.5078 -0.2880  0.4066 -0.3572
-0.3991  0.0816 -0.3843  0.6600  0.5008
 0.4564  0.0036 -0.4467 -0.4553  0.6204
-0.1622  0.8041  0.4480 -0.1725  0.3108
[torch.DoubleTensor of dimension 5x5]

> v * torch.diag(e:select(2,1))*v:t()
 1.9600 -6.4900 -0.4700 -7.2000 -0.6500
-6.4900  3.8000 -6.3900  1.5000 -6.3400
-0.4700 -6.3900  4.1700 -1.5100  2.6700
-7.2000  1.5000 -1.5100  5.7000  1.8000
-0.6500 -6.3400  2.6700  1.8000 -7.1000
[torch.DoubleTensor of dimension 5x5]

> b:dist(v * torch.diag(e:select(2,1)) * v:t())
3.5423944346685e-14
```

torch.svd([resu, ress, resv,] a [, 'S' or 'A'])

`U,S,V=torch.svd(A)`는 `A = USV'*`인 한 `n x m` 실수 행렬 `A`의 특잇값 분해를 리턴합니다.

`U`는 `n x n`이고, `S`는 `n x m`이고, `V`는 `m x m`입니다.

마지막 인자(만약 그것이 문자열이면)는 계산될 특잇값들의 개수를 나타냅니다. `'S'`는 약간(*some*)을 나타내고 `'A'`는 모두(*all*)를 나타냅니다.

주의: 원래 strides를 개의치 않고, 리턴된 행렬 `u`는 전치됩니다. 다시 말해, strides로 `n,1` 대신 `1,n`을 가집니다.

```
> a = torch.Tensor({{8.79,  6.11, -9.15,  9.57, -3.49,  9.84},
                    {9.93,  6.91, -7.93,  1.64,  4.02,  0.15},
                    {9.83,  5.04,  4.86,  8.83,  9.80, -8.99},
                    {5.45, -0.27,  4.85,  0.74, 10.00, -6.02},
                    {3.16,  7.98,  3.01,  5.80,  4.27, -5.31}}):t()

> a
  8.7900   9.9300   9.8300   5.4500   3.1600
  6.1100   6.9100   5.0400  -0.2700   7.9800
 -9.1500  -7.9300   4.8600   4.8500   3.0100
  9.5700   1.6400   8.8300   0.7400   5.8000
 -3.4900   4.0200   9.8000  10.0000   4.2700
  9.8400   0.1500  -8.9900  -6.0200  -5.3100

> u,s,v = torch.svd(a)
> u
-0.5911  0.2632  0.3554  0.3143  0.2299
-0.3976  0.2438 -0.2224 -0.7535 -0.3636
-0.0335 -0.6003 -0.4508  0.2334 -0.3055
-0.4297  0.2362 -0.6859  0.3319  0.1649
-0.4697 -0.3509  0.3874  0.1587 -0.5183
 0.2934  0.5763 -0.0209  0.3791 -0.6526
[torch.DoubleTensor of dimension 6x5]

> s
 27.4687
 22.6432
  8.5584
  5.9857
  2.0149
[torch.DoubleTensor of dimension 5]

> v
-0.2514  0.8148 -0.2606  0.3967 -0.2180
-0.3968  0.3587  0.7008 -0.4507  0.1402
-0.6922 -0.2489 -0.2208  0.2513  0.5891
-0.3662 -0.3686  0.3859  0.4342 -0.6265
-0.4076 -0.0980 -0.4933 -0.6227 -0.4396
[torch.DoubleTensor of dimension 5x5]

> u * torch.diag(s) * v:t()
  8.7900   9.9300   9.8300   5.4500   3.1600
  6.1100   6.9100   5.0400  -0.2700   7.9800
 -9.1500  -7.9300   4.8600   4.8500   3.0100
  9.5700   1.6400   8.8300   0.7400   5.8000
 -3.4900   4.0200   9.8000  10.0000   4.2700
  9.8400   0.1500  -8.9900  -6.0200  -5.3100
[torch.DoubleTensor of dimension 6x5]
```

```
> a:dist(u * torch.diag(s) * v:t())
2.8923773593204e-14
```

torch.inverse([res,] x)

정방 행렬 x의 역행렬을 계산합니다.

=torch.inverse(x)는 그 결과를 새 텐서 하나로 리턴합니다.

torch.inverse(y,x)는 그 결과를 y에 넣습니다.

주의: 원래 strides를 개의치 않고, 리턴된 행렬 y는 전치됩니다. 다시 말해, strides로 m,1 대신 1,m을 가집니다.

```
> x = torch.rand(10,10)
> y = torch.inverse(x)
> z = x * y
> z
 1.0000 -0.0000  0.0000 -0.0000  0.0000  0.0000  0.0000 -0.0000  0.0000  0.0000
 0.0000  1.0000 -0.0000 -0.0000  0.0000  0.0000 -0.0000 -0.0000 -0.0000  0.0000
 0.0000 -0.0000  1.0000 -0.0000  0.0000  0.0000 -0.0000 -0.0000  0.0000  0.0000
 0.0000 -0.0000 -0.0000  1.0000 -0.0000  0.0000  0.0000 -0.0000 -0.0000  0.0000
 0.0000 -0.0000  0.0000 -0.0000  1.0000  0.0000  0.0000 -0.0000 -0.0000  0.0000
 0.0000 -0.0000  0.0000 -0.0000  0.0000  1.0000  0.0000 -0.0000 -0.0000  0.0000
 0.0000 -0.0000  0.0000 -0.0000  0.0000  0.0000  1.0000 -0.0000  0.0000  0.0000
 0.0000 -0.0000 -0.0000 -0.0000  0.0000  0.0000  0.0000  1.0000  0.0000  0.0000
 0.0000 -0.0000 -0.0000 -0.0000  0.0000  0.0000 -0.0000 -0.0000  1.0000  0.0000
 0.0000 -0.0000  0.0000 -0.0000  0.0000  0.0000  0.0000 -0.0000  0.0000  1.0000
[torch.DoubleTensor of dimension 10x10]

> torch.max(torch.abs(z- torch.eye(10))) -- 0이 아닌 최댓값
2.3092638912203e-14
```

torch.qr([q, r], x)

행렬 x의 QR 분해를 계산합니다. x = q * r인 행렬 q와 r에서, q는 직교 행렬이고 r은 상삼각 행렬입니다. 이 함수는 thin (reduced) QR 분해를 리턴합니다.

=torch.qr(x)는 Q와 R 요소들을 새 행렬들로 리턴합니다.

`torch.qr(q, r, x)`는 그 결과들을 이미 존재하는 텐서 `q`와 `r`에 저장합니다.

주의하십시오. 만약 `x`에 있는 요소들의 크기가 크면, 정밀도를 잃을 수도 있습니다.

또한, 주의하십시오. 이 함수가 항상 유효한 분해를 리턴하긴 하지만, 플랫폼이 달라지면 (유효하지만 살짝) 다른 결과를 리턴할 수도 있습니다. 이는 그 플랫폼에 설치된 라팩(LAPACK) 구현에 따라 달라집니다.

주의: 원래 strides를 개의치 않고, 리턴된 행렬 `q`는 전치됩니다. 다시 말해, strides로 `m,1` 대신 `1,m`을 가집니다.

```
> a = torch.Tensor{{12, -51, 4}, {6, 167, -68}, {-4, 24, -41}}
> =a
  12  -51    4
   6  167  -68
  -4   24  -41
[torch.DoubleTensor of dimension 3x3]

> q, r = torch.qr(a)
> =q
-0.8571  0.3943  0.3314
-0.4286 -0.9029 -0.0343
 0.2857 -0.1714  0.9429
[torch.DoubleTensor of dimension 3x3]

> =r
 -14.0000  -21.0000   14.0000
   0.0000 -175.0000   70.0000
   0.0000    0.0000  -35.0000
[torch.DoubleTensor of dimension 3x3]

> =(q*r):round()
  12  -51    4
   6  167  -68
  -4   24  -41
[torch.DoubleTensor of dimension 3x3]

> =(q:t()*q):round()
 1  0  0
 0  1  0
 0  0  1
[torch.DoubleTensor of dimension 3x3]
```

torch.geqrf([m, tau], a)

이 함수는 라팩(LAPACK)을 호출하기 위한 저수준 함수입니다. 보통은 이 함수 대신 `torch.qr()`이 사용됩니다.

`a`의 QR 분해를 계산합니다. 그러나 Q와 R을 명시적으로 분리된 행렬로 만들지는 않습니다. 오히려, 이 함수는 직접적으로 기저의 라팩 함수 `?geqrf`를 호출합니다. 이 함수는 'elementary reflectors'의 한 시퀀스를 생성합니다. 더 자세한 사항은 라팩 문서를 보십시오.

torch.orgqr([q], m, tau)

이 함수는 라팩(LAPACK)을 호출하기 위한 저수준 함수입니다. 보통은 이 함수 대신 `torch.qr()`이 사용됩니다.

`torch.geqrf`로 주어진 것과 같은 elementary reflectors에서 Q 행렬 하나를 생성합니다. 더 자세한 사항은 라팩 문서를 보십시오.

torch.ormqr([res], m, tau, mat [, 'L' or 'R'] [, 'N' or 'T'])

`geqrf`에 의해 리턴된 elementary reflectors와 스칼라 요소(factor)들로 정의된 Q를 한 행렬과 곱합니다. 이 함수는 라팩(LAPACK)을 호출하기 위한 저수준 함수입니다. 보통은 이 함수 대신 `torch.qr()`이 사용됩니다.

- `side`(`'L'` 또는 `'R'`)는 `mat`가 왼쪽에 곱해져야 하는지(`mat * Q`) 오른쪽에 곱해져야 하는지(`Q * mat`)를 특정합니다.
- `trans`(`'N'` 또는 `'T'`)는 `Q`가 곱해지기 전에 전치되어야 하는지를 특정합니다.

더 자세한 사항은 라팩 문서를 보십시오.

텐서에 대한 논리 연산자

이 함수들은 논리 비교 연산자들을 구현합니다. 그 연산자들은 텐서 하나를 입력 받아서 그것을 또다른 텐서 하나 또는 숫자 하나와 비교합니다. 이 연산자들은 `ByteTensor` 하나를 리턴합니다. 그 `ByteTensor`의 각 요소는 각 비교 결과가 `false`였으면 0으로, `true`였으면 1로 채워집니다.

torch.lt(a, b)

`<` 연산자를 구현합니다. 만약 `b`가 숫자 하나이면, `<` 연산자는 `a`에 있는 각 요소를 `b`와 비교합니다. 그렇지 않으면, `<` 연산자는 `a`에 있는 각 요소를 그에 상응하는 `b`에 있는 요소와 비교합니다.

torch.le(a, b)

`<=` 연산자를 구현합니다. 만약 `b`가 숫자 하나이면, `<=` 연산자는 `a`에 있는 각 요소를 `b`와 비교합니다. 그렇지 않으면, `<=` 연산자는 `a`에 있는 각 요소를 그에 상응하는 `b`에 있는 요소와 비교합니다.

torch.gt(a, b)

`>` 연산자를 구현합니다. 만약 `b`가 숫자 하나이면, `>` 연산자는 `a`에 있는 각 요소를 `b`와 비교합니다. 그렇지 않으면, `>` 연산자는 `a`에 있는 각 요소를 그에 상응하는 `b`에 있는 요소와 비교합니다.

torch.ge(a, b)

`>=` 연산자를 구현합니다. 만약 `b`가 숫자 하나이면, `>=` 연산자는 `a`에 있는 각 요소를 `b`와 비교합니다. 그렇지 않으면, `>=` 연산자는 `a`에 있는 각 요소를 그에 상응하는 `b`에 있는 요소와 비교합니다.

torch.eq(a, b)

`==` 연산자를 구현합니다. 만약 `b`가 숫자 하나이면, `==` 연산자는 `a`에 있는 각 요소를 `b`와 비교합니다. 그렇지 않으면, `==` 연산자는 `a`에 있는 각 요소를 그에 상응하는 `b`에 있는 요소와 비교합니다.

torch.ne(a, b)

`!=` 연산자를 구현합니다. 만약 `b`가 숫자 하나이면, `!=` 연산자는 `a`에 있는 각 요소를 `b`와 비교합니다. 그렇지 않으면, `!=` 연산자는 `a`에 있는 각 요소를 그에 상응하는 `b`에 있는 요소와 비교합니다.

torch.all(a)

`ByteTensor` `a`에 있는 모든 요소가 `0`이 아니거나 `true`이면 `true`를 리턴합니다.

torch.any(a)

ByteTensor a에 있는 요소 중 하나라도 0이 아니거나 true이면 true를 리턴합니다.

```
> a = torch.rand(10)
> b = torch.rand(10)
> a
 0.5694
 0.5264
 0.3041
 0.4159
 0.1677
 0.7964
 0.0257
 0.2093
 0.6564
 0.0740
[torch.DoubleTensor of dimension 10]

> b
 0.2950
 0.4867
 0.9133
 0.1291
 0.1811
 0.3921
 0.7750
 0.3259
 0.2263
 0.1737
[torch.DoubleTensor of dimension 10]

> torch.lt(a,b)
 0
 0
 1
 0
 1
 0
 1
 1
 0
 1
[torch.ByteTensor of dimension 10]

> torch.eq(a,b)
0
0
0
0
0
0
0
0
0
0
```

```
[torch.ByteTensor of dimension 10]

> torch.ne(a,b)
 1
 1
 1
 1
 1
 1
 1
 1
 1
 1
[torch.ByteTensor of dimension 10]

> torch.gt(a,b)
 1
 1
 0
 1
 0
 1
 0
 0
 1
 0
[torch.ByteTensor of dimension 10]

> a[torch.gt(a,b)] = 10
> a
 10.0000
 10.0000
  0.3041
 10.0000
  0.1677
 10.0000
  0.0257
  0.2093
 10.0000
  0.0740
[torch.DoubleTensor of dimension 10]

> a[torch.gt(a,1)] = -1
> a
-1.0000
-1.0000
 0.3041
-1.0000
 0.1677
-1.0000
 0.0257
 0.2093
-1.0000
 0.0740
[torch.DoubleTensor of dimension 10]
```

```
> a = torch.ones(3):byte()
> torch.all(a)
true

> a[2] = 0
> torch.all(a)
false

> torch.any(a)
true

> a:zero()
> torch.any(a)
false
```

목차

- [res] torch.atan([res,] x)
- [res] torch.ceil([res,] x)
- [res] torch.cos([res,] x)
- [res] torch.cosh([res,] x)
- [res] torch.exp([res,] x)
- [res] torch.floor([res,] x)
- [res] torch.log([res,] x)
- [res] torch.log1p([res,] x)
- x:neg()
- [res] torch.pow([res,] x, n)
- [res] torch.round([res,] x)
- [res] torch.sin([res,] x)
- [res] torch.sinh([res,] x)
- [res] torch.sqrt([res,] x)
- [res] torch.tan([res,] x)
- [res] torch.tanh([res,] x)

기본적인 연산

- [res] torch.add([res,] tensor, value)
- [res] torch.add([res,] tensor1, tensor2)
- [res] torch.add([res,] tensor1, value, tensor2)
- tensor:csub(value)
- tensor1:csub(tensor2)
- [res] torch.mul([res,] tensor1, value)
- [res] torch.clamp([res,] tensor, min_value, max_value)
- [res] torch.cmul([res,] tensor1, tensor2)
- [res] torch.cpow([res,] tensor1, tensor2)
- [res] torch.addcmul([res,] x [,value], tensor1, tensor2)
- [res] torch.div([res,] tensor, value)
- [res] torch.cdiv([res,] tensor1, tensor2)
- [res] torch.addcdiv([res,] x [,value], tensor1, tensor2)
- [number] torch.dot(tensor1,tensor2)
- [res] torch.addmv([res,] [beta,] [v1,] vec1, [v2,] mat, vec2)
- [res] torch.addr([res,] [v1,] mat, [v2,] vec1, vec2)
- [res] torch.addmm([res,] [beta,] [v1,] M [v2,] mat1, mat2)
- [res] torch.addbmm([res,] [v1,] M [v2,] batch1, batch2)
- [res] torch.baddbmm([res,] [v1,] M [v2,] batch1, batch2)
- [res] torch.mv([res,] mat, vec)
- [res] torch.mm([res,] mat1, mat2)
- [res] torch.bmm([res,] batch1, batch2)
- [res] torch.ger([res,] vec1, vec2)

오버로드된 연산자

❖ 틀렸거나 보완할 점을 본문에 댓글로 또는 저에게 이메일로 알려 주시면 감사하겠습니다.