텐서

2015년 10월 8일에 한국어로 옮겨짐
2015년 10월 27일에 마지막으로 새로 고쳐짐

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

텐서(Tensor) 클래스는 아마 토치(Torch)에서 가장 중요한 클래스일 것입니다. 거의 모든 패키지가 이 클래스에 의존합니다. 텐서 클래스에서 숫자 데이터가 다뤄집니다. 토치(Torch7)의 거의 모든 클래스처럼 텐서는 직렬화(serializable)될 수 있습니다.

다차원 행렬

텐서는 잠재적 다차원 행렬입니다. 차원의 개수는 무제한입니다. 롱스토리지를 사용하여 4차원 이상의 더 높은 차원도 만들 수 있습니다.

예:

```
--- 4차원 텐서 생성 4x5x6x2
z = torch.Tensor(4,5,6,2)
--- 더 많은 차원(여기서는 6차원 텐서)은 이렇게 만들 수 있습니다.
s = torch.LongStorage(6)
s[1] = 4; s[2] = 5; s[3] = 6; s[4] = 2; s[5] = 7; s[6] = 3;
x = torch.Tensor(s)
```

nDimension() 또는 dim()은 한 `Tensor`에 있는 차원의 개수를 리턴합니다. size(i)는 `i` 번째 차원의 크기를 리턴합니다. size()는 한 텐서를 구성하고 있는 각 차원의 크기를 리턴합니다.

```
> x:nDimension()
6
> x:size()
 4
 5
 6
 2
 7
 3
[torch.LongStorage of size 6]
```

내부적 데이터 표현 (Internal data representation)

텐서의 실제 데이터는 스토리지에 담깁니다. 그 스토리지에는 `storage()`로 접근합니다. 이 고유한 스토리지에는 텐서의 메모리가 저장됩니다. 그러나 스토리지는 연속적이 아닐 수도 있습니다. 그 스토리지에서 사용되는 (1로 시작하는) 첫 위치는 `storageOffset()`으로 얻을 수 있습니다. `stride(i)`는 `i` 번째 차원에 있는 한 요소에서 다른 한 요소로 이동하는 데 필요한 간격을 리턴합니다. 다시 말해, 3차원 텐서 하나가 있다면,

```
x = torch.Tensor(7,7,7)
```

요소 (3,4,5)는 다음과 같이 접근될 수 있습니다.

```
> x[3][4][5]
```

또한, 다음과 같이 결과는 같지만 느린 방법도 있습니다.

```
> x:storage()[x:storageOffset()
              +(3-1)*x:stride(1)+(4-1)*x:stride(2)+(5-1)*x:stride(3)]
```

텐서는 스토리지를 보는 하나의 특별한 방식이라고 말할 수도 있습니다. 스토리지 하나는 오직 한 메모리 덩어리(chunk)만을 나타냅니다. 텐서는 그 메모리 덩어리를 차원을 가진 것으로 해석합니다.

```
x = torch.Tensor(4,5)
s = x:storage()
for i=1,s:size() do -- 그 스토리지를 가득 채움
  s[i] = i
end
> x -- s는 x에 의해 이차원 행렬로 해석됩니다.
  1   2   3   4   5
  6   7   8   9  10
 11  12  13  14  15
 16  17  18  19  20
[torch.DoubleTensor of dimension 4x5]
```

주의하십시오. 토치(Torch7)의 한 행렬[텐서]에서는 **같은 행에 있는 요소들[마지막** 차원 방향으로 나란히 있는 요소들]이 메모리에서 연속적입니다.

```
x = torch.Tensor(4,5)
i = 0

x:apply(function()
  i = i + 1
  return i
end)

> x
  1   2   3   4   5
  6   7   8   9  10
 11  12  13  14  15
 16  17  18  19  20
[torch.DoubleTensor of dimension 4x5]

> x:stride()
 5
 1  -- 마지막 차원에 있는 요소는 연속적입니다!
[torch.LongStorage of size 2]
```

이는 정확히 C와 같고, 포트란(Fortran)과는 다릅니다.

텐서의 타입(type)

텐서에는 몇 가지 타입이 있습니다.

```
ByteTensor -- 는 unsigned char를 담습니다.
CharTensor -- 는 signed char를 담습니다.
ShortTensor -- 는 short를 담습니다.
IntTensor -- 는 int를 담습니다.
FloatTensor -- 는 float를 담습니다.
DoubleTensor -- 는 double을 담습니다.
```

대부분의 수 연산자들이 오직 `FloatTensor`와 `DoubleTensor`로 구현됩니다. 다른 텐서 타입들도 메모리를 아끼고 싶을 때 유용합니다.

기본 텐서 타입

편의를 위해, `torch.Tensor`라는 별명(alias)이 제공됩니다. 이것은 사용자가 타입에 독립적인 스크립트를 작성할 수 있게합니다. 이것은 우리는 다음과 같이 한 번의 호출로 우리가 원하는 텐서 타입을 선택할 수 있게 합니다.

```
torch.setdefaulttensortype('torch.FloatTensor')
```

더 자세한 설명은 torch.setdefaulttensortype을 보십시오. 기본 별명은 `torch.DoubleTensor`입니다.

효율적인 메모리 관리

이 클래스에 있는 *모든* 텐서 연산은 어떤 메모리 복사도 만들지 않습니다. 이 모든 메소드들은 이미 존재하는 텐서들을 변형하거나, (이미 존재하는 텐서와) *같은 스토리지*를 참조하는 새 텐서 하나를 리턴합니다. 이 마술같은 동작은 stride()와 storageOffset()을 잘 사용하여 내부적으로 얻어집니다. 예:

```
x = torch.Tensor(5):zero()
> x
0
0
0
0
0
[torch.DoubleTensor of dimension 5]
> x:narrow(1, 2, 3):fill(1) -- narrow()는 x와 같은 스토리지를 참조하는 텐서를 리턴합니다.
> x
 0
 1
 1
 1
 0
[torch.Tensor of dimension 5]
```

만약 텐서를 실제로 복사해야 하면, copy() 메소드를 쓸 수 있습니다.

```
y = torch.Tensor(x:size()):copy(x)
```

또는 다음과 같은 편의 메소드도 있습니다.

```
y = x:clone()
```

우리는 이제 텐서를 위한 모든 메소드들을 설명합니다. 만약 텐서의 타입을 특정하고 싶다면, 단지 그 이름을 텐서(`Tensor`)에서 (`CharTensor` 같은) 텐서의 다른 형태로 바꾸면 됩니다.

텐서 생성자 (Tensor constructors)

텐서 생성자는 새 텐서 객체 하나를 만듭니다. 또한, 텐서 생성자는 선택적으로 새 메모리를 할당합니다. 기본적으로, 새로 할당된 메모리의 요소들은 초기화되지 않습니다. 따라서 새로 할당된 메모리 요소들에는 아마 임의의 숫자들이 들어있을 것입니다. 여기서, 새 텐서 하나를 생성하는 몇 가지 방법들을 소개합니다.

torch.Tensor()

빈 텐서 하나를 리턴합니다.

torch.Tensor(tensor)

인자로 입력된 텐서와 같은 스토리지를 참조하는 새 텐서 하나를 리턴합니다. 그 리턴되는 텐서는 인자로 입력된 텐서와 같은 사이즈(size), 스트라이드(stride), 그리고 스토리지 오프셋(storage offset)을 가집니다.

새로 만들어진 텐서는 이제 인자로 입력된 텐서와 같은 스토리지를 봅니다. 따라서, 새로 만들어진 텐서의 요소를 수정하면, 인자로 입력된 텐서의 요소도 함께 영향을 받습니다. 반대 경우도 마찬가지입니다. 메모리 복사는 생기지 않습니다!

```
x = torch.Tensor(2,5):fill(3.14)
> x
 3.1400  3.1400  3.1400  3.1400  3.1400
 3.1400  3.1400  3.1400  3.1400  3.1400
[torch.DoubleTensor of dimension 2x5]

y = torch.Tensor(x)
> y
 3.1400  3.1400  3.1400  3.1400  3.1400
 3.1400  3.1400  3.1400  3.1400  3.1400
[torch.DoubleTensor of dimension 2x5]

y:zero()
> x -- x의 요소들은 y의 요소들과 같습니다!
0 0 0 0 0
0 0 0 0 0
[torch.DoubleTensor of dimension 2x5]
```

torch.Tensor(sz1 [,sz2 [,sz3 [,sz4]]])

(최대 4차원의) 텐서 하나를 만듭니다. 그 텐서의 사이즈는 `sz1 x sz2 x sx3 x sz4`일 것입니다.

torch.Tensor(sizes, [strides])

임의의 수의 차원을 가진 텐서 하나를 만듭니다. 롱스토리지 `sizes`는 그 텐서를 구성하는 각 차원의 크기를 지정합니다. 선택적 인자인 롱스토리지 `strides`는 각 차원의 한 요소에서 그다음 요소로 이동하는 데 필요한 간격을 지정합니다. 물론, `sizes`와 `strides`의 요소 개수는 반드시 같아야 합니다. 만약 그렇지않거나, `strides`의 몇몇 요소가 *음수*이면, stride()는 그 텐서가 가능한 메모리에서 연속이 되도록 계산될 것입니다.

예를 들어, 4x4x3x2의 한 4차원 텐서 하나를 만듭니다.

```
x = torch.Tensor(torch.LongStorage({4,4,3,2}))
```

`strides`를 조작하다보면 다소 재미있는 결과도 나올 수 있습니다.

```
x = torch.Tensor(torch.LongStorage({4}), torch.LongStorage({0})):zero() -- 그 텐서를 0으로 만듭니다.
x[1] = 1 -- 모든 요소들이 같은 주소를 가리킵니다!
> x
 1
 1
 1
 1
[torch.DoubleTensor of dimension 4]
```

주의하십시오. *음수 strides는 허용되지 않습니다*. 만약 텐서를 만들 때 음수 strides가 인자로 주어지면, //그 텐서가 되도록 메모리에서 연속적이도록 stride를 선택하라는 뜻으로// 해석될 것입니다.

주의하십시오. 이 메소드는 *torch.LongTensor*들을 만드는 데 사용될 수 없습니다. 한 스토리지로부터의 생성자가 사용될 것입니다.

```
a = torch.LongStorage({1,2}) -- 값 1과 2를 가진 torch.LongStorage 하나가 있습니다.
-- 일반적인 경우 TYPE ~= Long, 이를테면 TYPE = Float:
b = torch.FloatTensor(a)
-- 1차원의 크기는 1이고 2차원의 크기는 2인 새 2차원 torch.FloatTensor 하나를 만듭니다.
> b:size()
 1
 2
[torch.LongStorage of size 2]

-- torch.LongTensor의 특별한 경우
c = torch.LongTensor(a)
-- 새로운 torch.LongTensor 하나를 만듭니다. 그 텐서는 그 스토리지를 사용합니다. 따라서 값 1과 2를 가집니다.
> c
```

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

torch.Tensor(storage, [storageOffset, sizes, [strides]])

(1보다 크거나 같은) `storageOffset` 위치에서 시작하는, 이미 존재하는 스토리지 `storage`를 사용하는 텐서 하나를 리턴합니다. 그 텐서의 각 차원은 롱스토리지 `sizes`를 참조하여 설정됩니다.

만약 인자로 `storage`가 입력되면, 생성자는 그 스토리지 전체를 보는 1차원 텐서 하나를 만들 것입니다.

각 차원에서 한 요소에서 다른 요소로 점프하는데 필요한 간격은 선택적 인자인 롱스토리지 `strides`로 입력됩니다. 만약 입력되지 않거나, 몇몇 요소들이 음수이면, stride()는 그 텐서가 메모리에서 가능한 연속적인 것처럼 계산될 것입니다.

`Storage`의 요소를 수정하면, 새로운 텐서의 요소에도 영향을 미칩니다. 반대 경우도 마찬가지입니다. 메모리 복사는 일어나지 않습니다!

```
-- 10개의 요소를 가진 스토리지 하나를 만듭니다.
s = torch.Storage(10):fill(1)

-- 우리는 그것을 2x5 텐서로 보길 바랍니다.
x = torch.Tensor(s, 1, torch.LongStorage{2,5})
> x
 1  1  1  1  1
 1  1  1  1  1
[torch.DoubleTensor of dimension 2x5]

x:zero()
> s -- 그 스토리지의 내용이 바뀌었습니다.
 0
 0
 0
 0
 0
 0
 0
 0
 0
 0
[torch.DoubleStorage of size 10]
```

torch.Tensor(storage, [storageOffset, sz1 [, st1 ... [, sz4 [, st4]]]])

(이전 생성자를 위한) 차원이 4보다 작거나 같다고 가정하는 편의 생성자. `szi`는 `i` 번째 차원의 크기이고, `sti`는 `i` 번째 차원의 스트라이드입니다.

torch.Tensor(table)

인자로 입력된 테이블은 루아 수들로 구성된 배열이라고 가정됩니다. 생성자는 그 테이블과 사이즈가 같고 요소들도 그대로 유지된 새 텐서 하나를 리턴합니다. 그 테이블은 다차원이었을 수도 있습니다.

예:

```
> torch.Tensor({{1,2,3,4}, {5,6,7,8}})
 1  2  3  4
 5  6  7  8
[torch.DoubleTensor of dimension 2x4]
```

함수 호출에 대한 노트

이 지침서의 나머지에서는 텐서들을 조작하는 여러 함수들을 소개할 것입니다. 대부분의 함수들은 유연하게 호출될 수 있게 정의되었습니다. 유연하다는 말은 객체 지향 "메소드 호출" 방식(이를테면 `src:function(...)`)으로 호출되거나 또는 더 "함수적인" 방식(`torch.function(src, ...)`)으로 호출될 수 있다는 뜻입니다. 여기서 `src`는 한 텐서입니다. 그러나 주의하십시오. 호출 방식은 그 결과를 호출한 텐서에 다시 저장할지, 아니면 새로운 텐서를 만들어 그곳에 저장할지에 따라 다를 수도 있습니다.

추가적으로, 몇몇 함수들은 `dst:function(src, ...)`의 형태로 호출될 수 있습니다. 이 형태는 보통 그 `src` 텐서에 대한 연산 결과가 텐서 `dst`에 저장됨을 암시합니다. 더 자세한 사항은 아래 각 함수의 정의에서 다뤄집니다.

그러나 주의하셔야 합니다. 제가 말씀드린 것에 관련하여 이 문서는 현재 불완전합니다. 저는 이 글을 읽으시는 분께서 직접 코드를 돌려보며 실험하시길 권장합니다.

복제 (Cloning)

[Tensor] clone()

한 텐서의 클론(복제품) 하나를 리턴합니다. 메모리 복사가 일어납니다.

```
i = 0
x = torch.Tensor(5):apply(function(x)
  i = i + 1
  return i
end)
> x
 1
 2
 3
 4
 5
[torch.DoubleTensor of dimension 5]

-- x의 클론 생성
y = x:clone()
> y
 1
 2
 3
 4
 5
[torch.DoubleTensor of dimension 5]

-- y를 1로 채움
y:fill(1)
> y
 1
 1
 1
 1
 1
[torch.DoubleTensor of dimension 5]

-- x의 내용은 바뀌지 않습니다.
> x
 1
 2
 3
 4
 5
[torch.DoubleTensor of dimension 5]
```

[Tensor] contiguous

- 만약 인자로 입력된 텐서의 내용이 메모리에서 연속적이면, 정확히 같은 텐서를 리턴합니다. (메모리 복사는 없습니다).
- 그렇지 않으면 (메모리에서 연속적이지 않으면), 클론 하나를 리턴합니다. (메모리 복사가 일어납니다).

```
x = torch.Tensor(2,3):fill(1)
> x
 1  1  1
 1  1  1
[torch.DoubleTensor of dimension 2x3]

-- x는 연속적입니다. 그래서 y는 정확히 같은 것을 가리킵니다.
y = x:contiguous():fill(2)
> y
 2  2  2
 2  2  2
[torch.DoubleTensor of dimension 2x3]

-- x의 내용도 바뀌었습니다.
> x
 2  2  2
 2  2  2
[torch.DoubleTensor of dimension 2x3]

-- x:t() 연속적이지 않습니다. 그래서 z는 클론입니다.
z = x:t():contiguous():fill(3.14)
> z
 3.1400  3.1400
 3.1400  3.1400
 3.1400  3.1400
[torch.DoubleTensor of dimension 3x2]

-- x의 내용은 바뀌지 않았습니다.
> x
 2  2  2
 2  2  2
[torch.DoubleTensor of dimension 2x3]
```

[Tensor or string] type(type)

만약 `type`이 `nil`이면, 인자로 입력된 텐서의 타입 이름을 나타내는 문자열을 리턴합니다.

```
= torch.Tensor():type()
torch.DoubleTensor
```

만약 `type`이 텐서 타입을 나타내는 **문자열이고**, 그 문자열이 인자로 입력된 텐서의 타입 이름과 같으면, 그 정확히 같은 텐서를 리턴합니다. (//메모리 복사는 생기지 않습니다//).

```
x = torch.Tensor(3):fill(3.14)
> x
 3.1400
 3.1400
 3.1400
[torch.DoubleTensor of dimension 3]

y = x:type('torch.DoubleTensor')
> y
 3.1400
 3.1400
 3.1400
[torch.DoubleTensor of dimension 3]

-- y의 내용을 0으로 만듭니다.
y:zero()

-- x의 내용이 바뀌었습니다.
> x
 0
 0
 0
[torch.DoubleTensor of dimension 3]
```

만약 `type`이 텐서 타입을 나타내는 **문자열이고**, 그 문자열이 인자로 입력된 텐서 타입 이름과 다르면, 그 특정된 타입의 새 텐서 하나를 리턴합니다. 그 새로 생긴 텐서의 내용은 원본 텐서와 같고, 타입만 인자로 입력된 타입으로 형 변환(cast)됩니다. (//메모리 복사가 일어나며, 정밀도 손실도 일어날 수 있습니다//).

```
x = torch.Tensor(3):fill(3.14)
> x
 3.1400
 3.1400
 3.1400
[torch.DoubleTensor of dimension 3]

y = x:type('torch.IntTensor')
> y
 3
 3
 3
[torch.IntTensor of dimension 3]
```

[Tensor] typeAs(tensor)

type 메소드를 위한 편의 메소드. 이 메소드의 동작은 다음과 같습니다.

```
type(tensor:type())
```

[boolean] isTensor(object)

인자로 입력된 `object`가 `torch.*Tensor` 타입 중 하나이면, `true`를 리턴합니다.

```
> torch.isTensor(torch.randn(3,4))
true

> torch.isTensor(torch.randn(3,4)[1])
true

> torch.isTensor(torch.randn(3,4)[1][2])
false
```

[Tensor] byte(), char(), short(), int(), long(), float(), double()

type 메소드를 위한 편의 메소드. 예를 들어,

```
x = torch.Tensor(3):fill(3.14)
> x
 3.1400
 3.1400
 3.1400
[torch.DoubleTensor of dimension 3]

-- type('torch.IntTensor') 호출
> x:type('torch.IntTensor')
 3
 3
 3
[torch.IntTensor of dimension 3]

-- 은 int()를 호출하는 것과 같습니다.
> x:int()
 3
 3
 3
[torch.IntTensor of dimension 3]
```

크기와 구조에 대한 문의 (Querying the size and structure)

[number] nDimension()

한 텐서가 몇 개의 차원으로 구성되어 있는지를 리턴합니다.

```
x = torch.Tensor(4,5) -- a matrix
> x:nDimension()
2
```

[number] dim()

nDimension()과 같습니다.

[number] size(dim)

특정된 차원 `dim`의 크기를 리턴합니다. 예:

```
x = torch.Tensor(4,5):zero()
> x
 0 0 0 0 0
 0 0 0 0 0
 0 0 0 0 0
 0 0 0 0 0
[torch.DoubleTensor of dimension 4x5]

> x:size(2) -- 열의 개수를 얻습니다.
5
```

[LongStorage] size()

그 텐서의 각 차원 크기를 포함하는 롱스토리지 하나를 리턴합니다.

```
x = torch.Tensor(4,5):zero()
> x
 0 0 0 0 0
```

```
 0 0 0 0 0
 0 0 0 0 0
 0 0 0 0 0
[torch.DoubleTensor of dimension 4x5]

> x:size()
 4
 5
[torch.LongStorage of size 2]
```

[LongStorage] #self

size() 메소드와 같습니다.

[number] stride(dim)

특정된 차원 `dim`을 위해, 한 요소에서 그다음 요소로 이동하기 위한 간격을 리턴합니다. 예:

```
x = torch.Tensor(4,5):zero()
> x
 0 0 0 0 0
 0 0 0 0 0
 0 0 0 0 0
 0 0 0 0 0
[torch.DoubleTensor of dimension 4x5]

-- 한 행에 있는 요소들은 메모리에서 연속적입니다.
> x:stride(2)
1

-- 한 열에 있는 요소에서 그다음 요소로 이동하려면,
-- 행 크기만큼 점프할 필요가 있습니다.
> x:stride(1)
5
```

주의하십시오. 토치에서, 한 행렬[텐서]의 *같은 행에 있는 요소들*[**마지막** 차원 방향으로 나란히 있는 요소들]이 메모리에서 연속적입니다.

[LongStorage] stride()

각 차원에서, 한 요소에서 그다음 요소로 이동하는 데 필요한 간격을 리턴합니다. 예:

```
x = torch.Tensor(4,5):zero()
> x
 0 0 0 0 0
 0 0 0 0 0
 0 0 0 0 0
 0 0 0 0 0
[torch.DoubleTensor of dimension 4x5]

> x:stride()
 5
 1 -- 한 행[마지막 차원]에서 요소들은 연속적입니다
[torch.LongStorage of size 2]
```

주의하십시오. 토치의 한 행렬[텐서]에서, *같은 행에 있는 요소들*[**마지막** 차원에 나란히 있는 요소들]이 메모리에서 연속적입니다.

[Storage] storage()

그 텐서의 모든 요소들을 저장했던 스토리지를 리턴합니다. 기본적으로, 텐서는 스토리지를 *보는* 특별한 방법의 하나입니다.

```
x = torch.Tensor(4,5)
s = x:storage()
for i=1,s:size() do -- 그 스토리지를 채웁니다.
  s[i] = i
end

> x -- s는 x에 의해 2차원 행렬로 해석됩니다.
  1   2   3   4   5
  6   7   8   9  10
 11  12  13  14  15
 16  17  18  19  20
[torch.DoubleTensor of dimension 4x5]
```

[boolean] isContiguous()

그 텐서의 요소들이 메모리에서 연속적이면 `true`를 리턴합니다.

```
-- 보통 텐서들은 메모리에서 연속적입니다.
x = torch.randn(4,5)
> x:isContiguous()
true

-- y는 이제 x의 세 번째 열을 "봅니다".
```

```
-- y의 스토리지는 x와 같습니다.
-- 따라서 메모리는 연속적일 수 없습니다.
y = x:select(2, 3)
> y:isContiguous()
false

-- 실제로, 한 요소에서 그다음 요소로 점프하기 위한 stride는 5입니다.
> y:stride()
 5
[torch.LongStorage of size 1]
```

[boolean] isSize(storage)

그 텐서의 차원들이 `storage`에 있는 요소들과 매치될 때 `true`를 리턴합니다.

```
x = torch.Tensor(4,5)
y = torch.LongStorage({4,5})
z = torch.LongStorage({5,4,1})
> x:isSize(y)
true

> x:isSize(z)
false

> x:isSize(x:size())
true
```

[boolean] isSameSizeAs(tensor)

이 메소드를 호출한 텐서와 인자로 입력된 텐서의 차원이 정확히 같으면 `true`을 리턴합니다.

```
x = torch.Tensor(4,5)
y = torch.Tensor(4,5)
> x:isSameSizeAs(y)
true

y = torch.Tensor(4,6)
> x:isSameSizeAs(y)
false
```

[number] nElement()

한 텐서에 있는 요소의 개수를 리턴합니다.

```
x = torch.Tensor(4,5)
> x:nElement() -- 4x5 = 20!
20
```

[number] storageOffset()

그 텐서의 스토리지에서 사용되는 (1로 시작하는) 첫 번째 인덱스(index)를 리턴합니다.

요소에 대한 문의 (Querying elements)

한 텐서의 요소들은 `[인덱스]` 연산자로 검색될 수 있습니다.

만약 `인덱스`가 숫자이면, `[인덱스]` 연산자는 `select(1, index)`와 같습니다. 만약 그 텐서가 2차원 이상이면, 이 연산은 같은 스토리지를 공유하는 그 텐서의 단면(slice) 하나를 리턴합니다. 만약 그 텐서가 일차원이면, `[인덱스]` 연산자는 그 텐서에서 `인덱스` 번째 값을 리턴합니다.

만약 `인덱스`가 테이블이면, 그 테이블은 반드시 *n* 개 숫자들을 포함해야 합니다. 여기서 *n*은 그 텐서가 가진 차원의 개수입니다. 그것은 그 주어진 위치에 있는 요소를 리턴할 것입니다.

같은 식으로, `인덱스`는 (그 텐서에서) 검색될 요소의 위치를 특정하는 롱스토리지일 수도 있습니다.

만약 `인덱스`가 0또는 1인 요소 구성된 `ByteTensor`이면, 그것은 원본 텐서의 한 부분 집합을 추출하기 위한 선택 마스크(selection mask)처럼 동작합니다. 그 마스크는 `torch.le` 같은 논리 연산자들과 함께 쓰일 때 매우 유용합니다.

예:

```
x = torch.Tensor(3,3)
i = 0; x:apply(function() i = i + 1; return i end)
> x
 1  2  3
 4  5  6
 7  8  9
[torch.DoubleTensor of dimension 3x3]

> x[2] -- 행 2를 리턴합니다.
 4
```

```
 5
 6
[torch.DoubleTensor of dimension 3]

> x[2][3] -- 2행 3열을 리턴합니다.
6

> x[{2,3}] -- 2행 3열을 리턴하는 또다른 방법입니다.
6

> x[torch.LongStorage{2,3}] -- 2행 3열을 리턴하는 또다른 방법입니다.
6

> x[torch.le(x,3)] -- torch.le는 마스크처럼 동작하는 한 ByteTensor를 리턴합니다.
 1
 2
 3
[torch.DoubleTensor of dimension 3]
```

텐서를 이미 존재하는 텐서 또는 메모리 덩어리를 통해 참조하기 (Referencing a tensor to an existing tensor or chunk of memory)

텐서는 스토리지를 보는 한 가지 방법입니다. 텐서는 이미 존재하는 스토리지를 보도록 "설정"될 수 있습니다.

유념하십시오. 만약 당신이 다음과 같이 빈 텐서를 설정하고 싶다면,

```
y = torch.Storage(10)
x = torch.Tensor()
x:set(y, 1, 10)
```

다음과 같이 대등한 생성자 중 하나를 사용할 수도 있습니다.

```
y = torch.Storage(10)
x = torch.Tensor(y, 1, 10)
```

[self] set(tensor)

set()을 호출한 텐서는 이제 인자로 입력된 텐서와 같은 스토리지를 "볼" 것입니다. 따라서, set()을 호출한 텐서에 있는 요소를 바꾸면, 인자로 입력된 텐서의 요소도 바뀝니다. 반대 경우도 마찬가지입니다. 이 메소드는 메모리 복사가 없는 효율적인 메소드입니다.

```
x = torch.Tensor(2,5):fill(3.14)
> x
 3.1400  3.1400  3.1400  3.1400  3.1400
 3.1400  3.1400  3.1400  3.1400  3.1400
[torch.DoubleTensor of dimension 2x5]

y = torch.Tensor():set(x)
> y
 3.1400  3.1400  3.1400  3.1400  3.1400
 3.1400  3.1400  3.1400  3.1400  3.1400
[torch.DoubleTensor of dimension 2x5]

y:zero()
> x -- x의 요소들은 y와 같습니다!
 0 0 0 0 0
 0 0 0 0 0
[torch.DoubleTensor of dimension 2x5]
```

[self] set(storage, [storageOffset, sizes, [strides]])

set()을 호출한 텐서는 이제 인자로 입력된 스토리지를 볼 것입니다. 그 스토리지는 (1보다 크거나 같은) `storageOffset`에서 시작합니다. 그 스토리지는 인자로 입력된 차원 `sizes`를 가집니다. 또한, 그 스토리지는, 선택적으로, 인자로 입력된 `strides`를 가집니다. 따라서, 그 스토리지의 요소를 수정하면, 그 텐서의 요소도 바뀝니다. 반대 경우도 마찬가지입니다. 이 메소드는 메모리 복사가 없는 효율적인 메소드입니다!

`storage`가 인자로 입력된 경우에만, 전체 스토리지를 1차원 텐서로 볼 수 있습니다.

```
-- 10개 요소를 가진 스토리지 하나를 만듭니다.
s = torch.Storage(10):fill(1)

-- 우리는 그것을 2x5 텐서로 보고자 합니다.
sz = torch.LongStorage({2,5})
x = torch.Tensor()
x:set(s, 1, sz)
> x
 1  1  1  1  1
 1  1  1  1  1
[torch.DoubleTensor of dimension 2x5]
```

```
x:zero()
> s -- 스토리지 내용이 바뀌었습니다.
 0
 0
 0
 0
 0
 0
 0
 0
 0
 0
[torch.DoubleStorage of size 10]
```

[self] set(storage, [storageOffset, sz1 [, st1 ... [, sz4 [, st4]]]])

이것은 이전 메소드의 한 간단한 버전입니다. 이 함수는 4차원까지만 동작합니다. `szi`는 그 텐서의 `i` 번째 차원의 크기입니다. `sti`는 `i` 번째 차원의 스트라이드입니다.

복사 및 초기화 (Copying and initializing)

[self] copy(tensor)

copy()를 호출한 텐서의 요소들을 인자로 넣은 텐서의 요소들로 바꿉니다. 두 텐서의 요소 개수는 반드시 같아야합니다. 그러나 차원(sizes)은 다를 수도 있습니다.

```
x = torch.Tensor(4):fill(1)
y = torch.Tensor(2,2):copy(x)
> x
 1
 1
 1
 1
[torch.DoubleTensor of dimension 4]

> y
 1  1
 1  1
[torch.DoubleTensor of dimension 2x2]
```

만약 다른 타입의 텐서가 주어지면, 형 변환이 일어납니다. 물론, 이것은 정밀도 손실을 일으킬 수도 있습니다.

[self] fill(value)

fill()을 호출한 텐서의 요소들을 인자로 입력된 값(value)으로 채웁니다.

```
> torch.DoubleTensor(4):fill(3.14)
 3.1400
 3.1400
 3.1400
 3.1400
[torch.DoubleTensor of dimension 4]
```

[self] zero()

zero()를 호출한 텐서의 요소들을 0으로 채웁니다.

```
> torch.Tensor(4):zero()
 0
 0
 0
 0
[torch.DoubleTensor of dimension 4]
```

크기 바꾸기 (Resizing)

더 큰 크기로 바꿀 때, 그 기저의 스토리지는 텐서의 모든 요소들에 맞추기 위해 크기가 바뀝니다.
더 작은 크기로 바꿀 때, 그 기저의 스토리지 크기는 바뀌지 않습니다.
중요: 크기를 바꾼 뒤 텐서의 내용은 *정해지지 않습니다*. 스트라이드가 완전히 바뀌었을 수도 있기 때문입니다. 특히, *그 크기가 바뀐 텐서의 요소들은 메모리에서 연속적입니다*.

[self] resizeAs(tensor)

resizeAs()를 호출한 텐서의 크기를 (같은 타입의) 인자로 넣은 텐서의 크기로 바꿉니다.

[self] resize(sizes)

resize()를 호출한 텐서의 크기를 인자로 넣은 롱스토리지 `sizes`로 바꿉니다.

[self] resize(sz1 [,sz2 [,sz3 [,sz4]]])

최대 4차원까지 동작하는 resize(sizes)의 편의 메소드.

서브텐서 추출 (Extracting sub-tensors)

이 각각의 메소드들은 텐서 하나를 리턴합니다. 그 리턴되는 텐서는 *같은 스토리지를 가진* 인자로 입력된 텐서의 서브텐서입니다. 그러므로, 그 서브텐서의 메모리 내용을 바꾸면, 그 상위 텐서에도 영향을 미칩니다. 반대 경우도 마찬가지입니다.

이 메소드들에는 어떠한 메모리 복사도 없기 때문에, 매우 빠릅니다.

[self] narrow(dim, index, size)

현재 텐서에서 특정 부분을 추출한 새로운 텐서 하나를 리턴합니다. 그 추출할 부분의 차원은 인자 `dim`으로 설정됩니다. 그 설정된 차원에서 추출할 범위는 `index`부터 `index+size-1`까지로 설정됩니다.

```
x = torch.Tensor(5, 6):zero()
> x

0 0 0 0 0 0
0 0 0 0 0 0
0 0 0 0 0 0
0 0 0 0 0 0
0 0 0 0 0 0
[torch.DoubleTensor of dimension 5x6]

y = x:narrow(1, 2, 3) -- 텐서 x의 차원 1에서, 인덱스 2부터 인덱스 2+3-1까지를 추출합니다.
y:fill(1) -- y를 1로 채웁니다.
> y
 1  1  1  1  1  1
 1  1  1  1  1  1
 1  1  1  1  1  1
[torch.DoubleTensor of dimension 3x6]

> x -- x의 메모리가 바뀌었습니다!
 0  0  0  0  0  0
 1  1  1  1  1  1
 1  1  1  1  1  1
```

```
 1  1  1  1  1  1
 0  0  0  0  0  0
[torch.DoubleTensor of dimension 5x6]
```

[Tensor] sub(dim1s, dim1e ... [, dim4s [, dim4e]])

이 메소드는 첫 네 차원을 차례로 narrow한 것과 같습니다. 이 메소드는 새로운 텐서 하나를 리턴합니다. 그 새로운 텐서는 `i` 번째 차원의 인덱스 `dimis`에서 `dimie`로 가는 한 서브텐서입니다. 음수 값들은 끝에서부터 시작하는 인덱스로 해석됩니다. 예를 들어, `-1`은 마지막 인덱스이고, `-2`는 마지막 인덱스보다 하나 전 인덱스입니다.

```
x = torch.Tensor(5, 6):zero()
> x
 0 0 0 0 0 0
 0 0 0 0 0 0
 0 0 0 0 0 0
 0 0 0 0 0 0
 0 0 0 0 0 0
[torch.DoubleTensor of dimension 5x6]

y = x:sub(2,4):fill(1) -- y는 x의 서브텐서입니다.
> y                    -- 차원 1은 인덱스 2에서 시작하여 인덱스 4에서 끝납니다.
 1  1  1  1  1  1
 1  1  1  1  1  1
 1  1  1  1  1  1
[torch.DoubleTensor of dimension 3x6]

> x                    -- x가 바뀌었습니다!
 0  0  0  0  0  0
 1  1  1  1  1  1
 1  1  1  1  1  1
 1  1  1  1  1  1
 0  0  0  0  0  0
[torch.DoubleTensor of dimension 5x6]

z = x:sub(2,4,3,4):fill(2) -- 새로운 서브텐서 하나를 얻습니다.
> z                        -- 차원 1은 인덱스 2에서 시작하여 인덱스 4에서 끝납니다.
                           -- 차원 2는 인덱스 3에서 시작하여 인덱스 4에서 끝납니다.
 2  2
 2  2
 2  2
[torch.DoubleTensor of dimension 3x2]

> x                        -- x가 바뀌었습니다.
 0  0  0  0  0  0
 1  1  2  2  1  1
 1  1  2  2  1  1
 1  1  2  2  1  1
```

```
 0  0  0  0  0  0
[torch.DoubleTensor of dimension 5x6]

> y                          -- y가 바뀌었습니다.
 1  1  2  2  1  1
 1  1  2  2  1  1
 1  1  2  2  1  1
[torch.DoubleTensor of dimension 3x6]

> y:sub(-1, -1, 3, 4)      -- 음수 값들 = 경계들
 2  2
[torch.DoubleTensor of dimension 1x2]
```

[Tensor] select(dim, index)

차원 `dim`에서 인자로 입력된 `index` 번째의 한 텐서 단면(slice)을 리턴합니다. 그 리턴된 텐서는 차원 `dim`이 제거된 한 차원 작은 차원을 가집니다. 따라서 1차원 텐서에는 `select()`를 쓸 수 없습니다.

주의하십시오. 첫 번째 차원을 "선택하는 것(selecting)"은 [] 연산자를 쓰는 것과 같습니다.

```
x = torch.Tensor(5,6):zero()
> x
 0 0 0 0 0 0
 0 0 0 0 0 0
 0 0 0 0 0 0
 0 0 0 0 0 0
 0 0 0 0 0 0
[torch.DoubleTensor of dimension 5x6]

y = x:select(1, 2):fill(2) -- 1 차원의 2 번째 인덱스(행 2)를 선택하고 그것을 2로 채웁니다.
> y
 2
 2
 2
 2
 2
 2
[torch.DoubleTensor of dimension 6]

> x
 0  0  0  0  0  0
 2  2  2  2  2  2
 0  0  0  0  0  0
 0  0  0  0  0  0
 0  0  0  0  0  0
[torch.DoubleTensor of dimension 5x6]

z = x:select(2,5):fill(5) -- 2차원의 5 번째 인덱스(5번째 열)을 선택하고 5로 채웁니다.
> z
```

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

> x
 0  0  0  0  5  0
 2  2  2  2  5  2
 0  0  0  0  5  0
 0  0  0  0  5  0
 0  0  0  0  5  0
[torch.DoubleTensor of dimension 5x6]
```

[Tensor] [{ dim1,dim2,... }] or [{ {dim1s,dim1e}, {dim2s,dim2e} }]

인덱싱 연산자 []는 간결하고 효율적 방식으로 narrow/sub와 select에 결합되어 쓰일 수 있습니다. 인덱싱 연산자 []는 또한 (서브)텐서들을 복사하고 채우는 데 쓰일 수 있습니다.

이 연산자는 0과 1 요소들로 구성된 `ByteTensor` 입력 마스크와 함께 동작합니다. 그 예로는 논리 연산자가 있습니다.

```
x = torch.Tensor(5, 6):zero()
> x
 0 0 0 0 0 0
 0 0 0 0 0 0
 0 0 0 0 0 0
 0 0 0 0 0 0
 0 0 0 0 0 0
[torch.DoubleTensor of dimension 5x6]

x[{ 1,3 }] = 1 -- 1행 3열 요소를 1로 설정
> x
 0  0  1  0  0  0
 0  0  0  0  0  0
 0  0  0  0  0  0
 0  0  0  0  0  0
 0  0  0  0  0  0
[torch.DoubleTensor of dimension 5x6]

x[{ 2,{2,4} }] = 2  -- 2행 2~4열 요소들을 2로 설정
> x
 0  0  1  0  0  0
 0  2  2  2  0  0
 0  0  0  0  0  0
 0  0  0  0  0  0
 0  0  0  0  0  0
[torch.DoubleTensor of dimension 5x6]
```

```
x[{ {},4 }] = -1 -- 4 번째 열 전체를 -1로 설정
> x
 0  0  1 -1  0  0
 0  2  2 -1  0  0
 0  0  0 -1  0  0
 0  0  0 -1  0  0
 0  0  0 -1  0  0
[torch.DoubleTensor of dimension 5x6]

x[{ {},2 }] = torch.range(1,5) -- 1~5 값을 갖는 1차원 텐서를 x의 2 번째 열 전체에 복사
> x

 0  1  1 -1  0  0
 0  2  2 -1  0  0
 0  3  0 -1  0  0
 0  4  0 -1  0  0
 0  5  0 -1  0  0
[torch.DoubleTensor of dimension 5x6]

x[torch.lt(x,0)] = -2 -- 모든 0보다 작은 요소들을 -2로 설정
> x

 0  1  1 -2  0  0
 0  2  2 -2  0  0
 0  3  0 -2  0  0
 0  4  0 -2  0  0
 0  5  0 -2  0  0
[torch.DoubleTensor of dimension 5x6]
```

[Tensor] index(dim, index)

index()는 차원 `dim`에서 `torch.LongTensor index`에 들어있는 텐서 단면들을 찾아 리턴합니다. 그 리턴된 텐서는 원래 텐서와 같은 차원의 개수를 가집니다. 그 리턴된 텐서는 원본 텐서와 **다른** 스토리지를 사용합니다. -- 그 결과를 이미 존재하는 한 텐서에 저장하는 아래 예제를 보십시오.

```
x = torch.rand(5,5)
> x
 0.8020  0.7246  0.1204  0.3419  0.4385
 0.0369  0.4158  0.0985  0.3024  0.8186
 0.2746  0.9362  0.2546  0.8586  0.6674
 0.7473  0.9028  0.1046  0.9085  0.6622
 0.1412  0.6784  0.1624  0.8113  0.3949
[torch.DoubleTensor of dimension 5x5]

y = x:index(1,torch.LongTensor{3,1})     -- 3 번째 행과 1 번째 행만 추출
> y
 0.2746  0.9362  0.2546  0.8586  0.6674
 0.8020  0.7246  0.1204  0.3419  0.4385
```

```
[torch.DoubleTensor of dimension 2x5]

y:fill(1)
> y
 1  1  1  1  1
 1  1  1  1  1
[torch.DoubleTensor of dimension 2x5]

> x                                     -- x의 내용은 바뀌지 않음
 0.8020  0.7246  0.1204  0.3419  0.4385
 0.0369  0.4158  0.0985  0.3024  0.8186
 0.2746  0.9362  0.2546  0.8586  0.6674
 0.7473  0.9028  0.1046  0.9085  0.6622
 0.1412  0.6784  0.1624  0.8113  0.3949
[torch.DoubleTensor of dimension 5x5]
```

주의하십시오. 명시적인 `index` 함수는 인덱싱 연산자 `[]`와 다릅니다. 인덱싱 연산자 `[]`는 연속된 select와 narrow 연산들을 위한 문법적 줄임 표현입니다. 그러므로 인덱싱 연산자 `[]`는 항상 원본 텐서와 같은 스토리지를 공유하는 새로운 뷰(view) 하나를 리턴합니다. 그러나 명시적 `index` 함수는 같은 스토리지를 사용할 수 없습니다.

`result:index(source, ...)`꼴로 그 결과를 이미 존재하는 텐서에 저장하는 것은 불가능합니다.

```
x = torch.rand(5,5)
> x
 0.8020  0.7246  0.1204  0.3419  0.4385
 0.0369  0.4158  0.0985  0.3024  0.8186
 0.2746  0.9362  0.2546  0.8586  0.6674
 0.7473  0.9028  0.1046  0.9085  0.6622
 0.1412  0.6784  0.1624  0.8113  0.3949
[torch.DoubleTensor of dimension 5x5]

y = torch.Tensor()
y:index(x,1,torch.LongTensor{3,1})
> y
 0.2746  0.9362  0.2546  0.8586  0.6674
 0.8020  0.7246  0.1204  0.3419  0.4385
[torch.DoubleTensor of dimension 2x5]
```

[Tensor] indexCopy(dim, index, tensor)

`tensor`의 요소들을 indexCopy()를 호출한 텐서로 복사합니다. 그 복사는 호출한 텐서의 `dim` 차원 `index` 번째 텐서 단면들로 수행됩니다. 복사될 내용의 차원과 복사할 곳의 차원은 반드시 같아야 합니다. 그렇지 않으면 에러가 생깁니다.

```
> x
 0.8020  0.7246  0.1204  0.3419  0.4385
 0.0369  0.4158  0.0985  0.3024  0.8186
 0.2746  0.9362  0.2546  0.8586  0.6674
 0.7473  0.9028  0.1046  0.9085  0.6622
 0.1412  0.6784  0.1624  0.8113  0.3949
[torch.DoubleTensor of dimension 5x5]

z=torch.Tensor(5,2)
z:select(2,1):fill(-1)
z:select(2,2):fill(-2)
> z
-1 -2
-1 -2
-1 -2
-1 -2
-1 -2
[torch.DoubleTensor of dimension 5x2]

x:indexCopy(2,torch.LongTensor{5,1},z)  -- z의 1열은 x의 5열로, z의 2열은 x의 1열로 복사됨
> x
-2.0000  0.7246  0.1204  0.3419 -1.0000
-2.0000  0.4158  0.0985  0.3024 -1.0000
-2.0000  0.9362  0.2546  0.8586 -1.0000
-2.0000  0.9028  0.1046  0.9085 -1.0000
-2.0000  0.6784  0.1624  0.8113 -1.0000
[torch.DoubleTensor of dimension 5x5]
```

[Tensor] indexFill(dim, index, val)

indexFill()을 호출한 텐서의 차원 `dim`의 `index` 번째 텐서 단면들을 값 `val`로 채웁니다.

```
x=torch.rand(5,5)
> x
 0.8414  0.4121  0.3934  0.5600  0.5403
 0.3029  0.2040  0.7893  0.6079  0.6334
 0.3743  0.1389  0.1573  0.1357  0.8460
 0.2838  0.9925  0.0076  0.7220  0.5185
 0.8739  0.6887  0.4271  0.0385  0.9116
[torch.DoubleTensor of dimension 5x5]

x:indexFill(2,torch.LongTensor{4,2},-10)  --x의 2 번째 차원의 4 번째와 2 번째 텐서 단면을 -10으로 채움.
> x
  0.8414 -10.0000   0.3934 -10.0000   0.5403
  0.3029 -10.0000   0.7893 -10.0000   0.6334
  0.3743 -10.0000   0.1573 -10.0000   0.8460
  0.2838 -10.0000   0.0076 -10.0000   0.5185
  0.8739 -10.0000   0.4271 -10.0000   0.9116
[torch.DoubleTensor of dimension 5x5]
```

[Tensor] gather(dim, index)

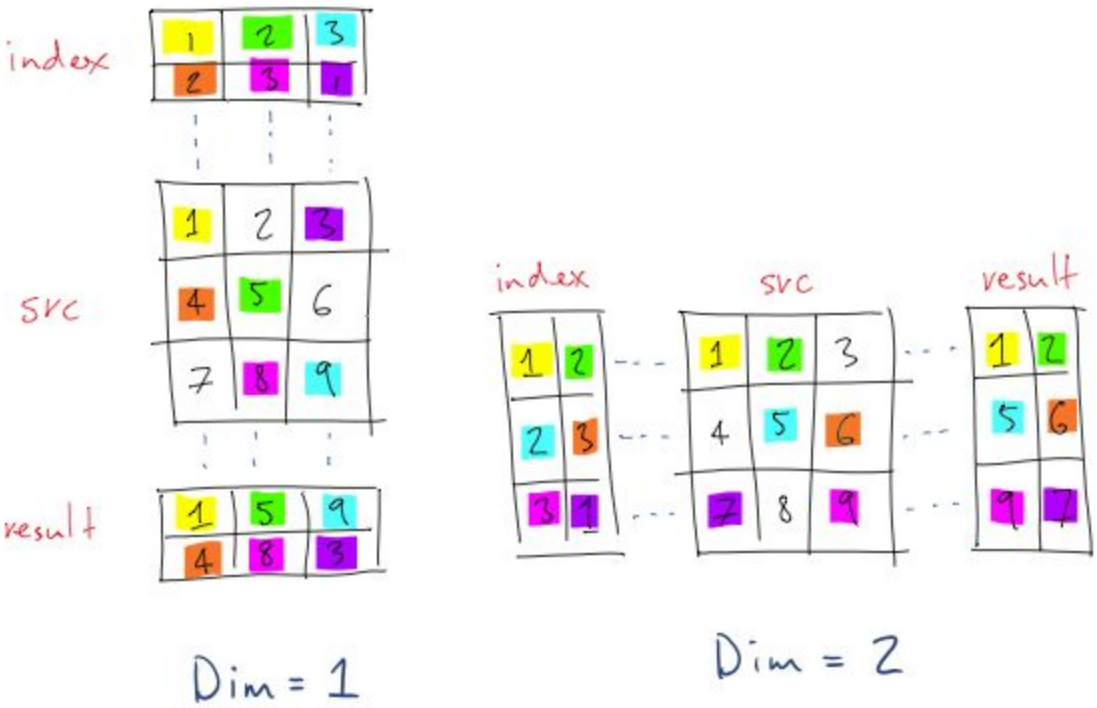
gather()를 호출하는 텐서를 `src` 텐서라 합시다. 우리 목표는 `src` 텐서에서 어떤 값들을 찾는 것입니다. 어떤 값을 찾을지는 `LongTensor index` 텐서로 정합니다. `index` 텐서의 각 요소는 `src` 텐서에서 몇 번째 값을 가져올 지를 나타냅니다. 차원 `dim`은 `index` 텐서를 `src` 텐서에 어떤 방향으로 적용할지를 결정합니다. 예를 들어, 만약 `dim`이 1이면, `index` 텐서는 `src` 텐서에 1 차원에 나란한 방향으로 적용됩니다. 결과 텐서는 `index` 텐서와 크기가 같습니다. 결과 텐서는 다음과 같이 주어집니다.

```
-- dim = 1
result[i][j][k]... = src[index[i][j][k]...][j][k]...

-- dim = 2
result[i][j][k]... = src[i][index[i][j][k]...][k]...

-- etc.
```

(`dim`이 2인 경우) `src` 텐서의 각 행에서 `index` 텐서의 각 행에 있는 요소 값 번째 값들이 선택됩니다. 같은 값은 한 행에서 한 번 이상 선택될 수 없습니다. `index` 텐서에 있는 요소 값들은 반드시 `src` 텐서의 한 행 길이보다는 작거나 같아야 합니다. 즉, `index` 텐서에 있는 요소 값들은 반드시 1에서 `src:size(dim)` 사이에 있어야 합니다. 그림으로 보면 다음과 같습니다.



수치적으로 예를 들면, 만약 `src`의 크기가 `n x m x p x q`이고, 우리가 `dim = 3`을 따라 수집하고 있으며, 우리가 (`k <= p`인) 각 행에서 `k` 개 요소들을 수집고자 한다면, `index` 텐서의 크기는 반드시 `n x m x k x q`이어야 합니다.

그 결과를 `result:gather(src, ...)` 꼴로 이미 존재하는 텐서에 저장하는 것도 가능합니다.

```
x = torch.rand(5, 5)
> x
 0.7259  0.5291  0.4559  0.4367  0.4133
 0.0513  0.4404  0.4741  0.0658  0.0653
 0.3393  0.1735  0.6439  0.1011  0.7923
 0.7606  0.5025  0.5706  0.7193  0.1572
 0.1720  0.3546  0.8354  0.8339  0.3025
[torch.DoubleTensor of size 5x5]

y = x:gather(1, torch.LongTensor{{1, 2, 3, 4, 5}, {2, 3, 4, 5, 1}})
> y
 0.7259  0.4404  0.6439  0.7193  0.3025
 0.0513  0.1735  0.5706  0.8339  0.4133
[torch.DoubleTensor of size 2x5]

z = x:gather(2, torch.LongTensor{{1, 2}, {2, 3}, {3, 4}, {4, 5}, {5, 1}})
> z
 0.7259  0.5291
 0.4404  0.4741
 0.6439  0.1011
 0.7193  0.1572
 0.3025  0.1720
[torch.DoubleTensor of size 5x2]
```

[Tensor] scatter(dim, index, src|val)

텐서 `src` 또는 스칼라 `val`에 있는 모든 값을 `self`의 특정된 인덱스들에 씁니다. 그 인덱스들은 차원 `dim` 방향으로 gather에서 설명한 방식 같이 결정됩니다. 유념하십시오. gather처럼 `index`의 값들은 반드시 1에서 `self:size(dim)` 사이에 있어야 합니다. 그 특정된 차원 방향에 있는 `index`의 모든 값들은 서로 겹치지 않아야 합니다.

```
x = torch.rand(2, 5)
> x
 0.3227  0.4294  0.8476  0.9414  0.1159
 0.7338  0.5185  0.2947  0.0578  0.1273
[torch.DoubleTensor of size 2x5]

y = torch.zeros(3, 5):scatter(1, torch.LongTensor{{1, 2, 3, 1, 1}, {3, 1, 1, 2, 3}}, x)
> y
 0.3227  0.5185  0.2947  0.9414  0.1159
 0.0000  0.4294  0.0000  0.0578  0.0000
```

```
 0.7338  0.0000  0.8476  0.0000  0.1273
[torch.DoubleTensor of size 3x5]

z = torch.zeros(2, 4):scatter(2, torch.LongTensor{{3}, {4}}, 1.23)
> z
 0.0000  0.0000  1.2300  0.0000
 0.0000  0.0000  0.0000  1.2300
[torch.DoubleTensor of size 2x4]
```

[Tensor] maskedSelect(mask)

마스크(mask)에 있는 요소 값 1에 대응되는 새 텐서 하나를 리턴합니다. mask는 각 요소가 0 또는 1로 구성된 torch.ByteTensor입니다. Tensor와 mask는 반드시 요소 개수가 같아야 합니다. 결과로 리턴될 텐서는 Tensor와 타입이 같고 크기는 mask:sum()인 1차원 텐서일 것입니다.

```
x = torch.range(1,12):double():resize(3,4)
> x
  1   2   3   4
  5   6   7   8
  9  10  11  12
[torch.DoubleTensor of dimension 3x4]

mask = torch.ByteTensor(2,6):bernoulli()
> mask
 1  0  1  0  0  0
 1  1  0  0  0  1
[torch.ByteTensor of dimension 2x6]

y = x:maskedSelect(mask)
> y
  1
  3
  7
  8
 12
[torch.DoubleTensor of dimension 5]

z = torch.DoubleTensor()
z:maskedSelect(x, mask)
> z
  1
  3
  7
  8
 12
```

x와 y의 차원은 다를 수도 있습니다. 이미 존재하는 텐서 z가 어떻게 결과를 저장하는 데 사용될 수 있는지도 보십시오.

[Tensor] maskedCopy(mask, tensor)

`tensor`의 마스크된 요소들을 그 자신으로 복사합니다. 마스크된 요소들은 `mask` 텐서의 1에 대응되는 요소들입니다. `mask`는 0 또는 1로 구성된 `torch.ByteTensor`입니다. 목적지 `Tensor`와 `mask` 텐서는 요소 개수가 꼭 같아야 합니다. 소스(source) `tensor`의 요소 개수는 최소한 `mask`에 있는 1의 개수보다는 많아야 합니다.

```
x = torch.range(1,4):double():resize(2,2)
> x
 1  2
 3  4
[torch.DoubleTensor of dimension 2x4]

mask = torch.ByteTensor(1,8):bernoulli()
> mask
 0  0  1  1  1  0  1  0
[torch.ByteTensor of dimension 1x8]

y = torch.DoubleTensor(2,4):fill(-1)
> y
-1 -1 -1 -1
-1 -1 -1 -1
[torch.DoubleTensor of dimension 2x4]

y:maskedCopy(mask, x)
> y
 -1 -1  1  2
  3 -1  4 -1
[torch.DoubleTensor of dimension 2x4]
```

x와 y의 차원은 다를 수도 있습니다. 그러나 요소 개수는 같아야 합니다.

[Tensor] maskedFill(mask, val)

그 자신의 마스크된 요소들을 값 `val`로 채웁니다. 마스크된 요소들은 `mask` 텐서에 있는 1에 대응되는 요소들입니다. `mask`는 0 또는 1로 구성된 `torch.ByteTensor`입니다. `mask`와 `Tensor`는 반드시 요소 개수가 같아야 합니다.

```
x = torch.range(1,4):double():resize(1,4)
> x
 1  2  3  4
```

```
[torch.DoubleTensor of dimension 1x4]

mask = torch.ByteTensor(2,2):bernoulli()
> mask
 0  0
 1  1
[torch.ByteTensor of dimension 2x2]

x:maskedFill(mask, -1)
> x
 1  2 -1 -1
[torch.DoubleTensor of dimension 1x4]
```

x와 mask의 차원은 다를 수도 있습니다. 그러나 요소 개수는 같아야 합니다.

검색 (Search)

이 각각의 메소드들은 주어진 검색 연산의 인덱스에 해당하는 LongTensor 하나를 리턴합니다.

[LongTensor] nonzero(tensor)

tensor에서 0이 아닌 요소들의 인덱스들을 LongTensor 하나에 담아 리턴합니다.

유념하십시오. 토치는 리턴 타입을 결정하기 위해 디스패치(dispatch)에서 첫 인자를 사용합니다. 첫 인자는 임의의 torch.TensorType일 수 있습니다. 그러나 리턴 타입은 항상 torch.LongTensor입니다. 그러므로, torch.nonzero(torch.LongTensor(), tensor)꼴로는 함수를 호출할 수 없습니다. 그러나 tensor.nonzero(torch.LongTensor(), tensor)는 동작합니다.

```
> x = torch.rand(4, 4):mul(3):floor():int()
> x
 2  0  2  0
 0  0  1  2
 0  2  2  1
 2  1  2  2
[torch.IntTensor of dimension 4x4]

> torch.nonzero(x)
 1  1
 1  3
 2  3
 2  4
 3  2
 3  3
 3  4
```

```
 4  1
 4  2
 4  3
 4  4
[torch.LongTensor of dimension 11x2]

> x:nonzero()
 1  1
 1  3
 2  3
 2  4
 3  2
 3  3
 3  4
 4  1
 4  2
 4  3
 4  4
[torch.LongTensor of dimension 11x2]

> indices = torch.LongTensor()
> x.nonzero(indices, x)
 1  1
 1  3
 2  3
 2  4
 3  2
 3  3
 3  4
 4  1
 4  2
 4  3
 4  4
[torch.LongTensor of dimension 11x2]

> x:eq(1):nonzero()
 2  3
 3  4
 4  2
[torch.LongTensor of dimension 3x2]
```

확장/복제/줄임 텐서들 (Expanding/Replicating/Squeezing Tensors)

이 메소드들은 원본 텐서를 복제하여 만들어진 텐서 하나를 리턴합니다.

[result] expand([result,] sizes)

`sizes`는 `torch.LongStorage` 한 개 또는 숫자 여러 개일 수 있습니다. 텐서의 확장은 메모리를 새로 할당하지 않습니다. 대신 이미 존재하는 텐서를 보는 새 뷰(view)를 만듭니다. 여기서 싱글턴 차원들은 `stride`를 0으로 설정함으로써 다차원으로 확장될 수 있습니다. 크기 1을 가진 어떤 차원도 임의의 값으로 확장될 수 있습니다. 이 때, 메모리 할당은 필요없습니다. 크기가 1이 아닌 차원을 따라 확장을 시도하면, 에러가 생길 것입니다.

```
x = torch.rand(10,1)
> x
 0.3837
 0.5966
 0.0763
 0.1896
 0.4958
 0.6841
 0.4038
 0.4068
 0.1502
 0.2239
[torch.DoubleTensor of dimension 10x1]

y = torch.expand(x,10,2)
> y
 0.3837  0.3837
 0.5966  0.5966
 0.0763  0.0763
 0.1896  0.1896
 0.4958  0.4958
 0.6841  0.6841
 0.4038  0.4038
 0.4068  0.4068
 0.1502  0.1502
 0.2239  0.2239
[torch.DoubleTensor of dimension 10x2]

y:fill(1)
> y
 1  1
 1  1
 1  1
 1  1
 1  1
 1  1
 1  1
 1  1
 1  1
 1  1
[torch.DoubleTensor of dimension 10x2]

> x
 1
 1
 1
 1
```

```
 1
 1
 1
 1
 1
 1
[torch.DoubleTensor of dimension 10x1]

i=0; y:apply(function() i=i+1;return i end)
> y
  2   2
  4   4
  6   6
  8   8
 10  10
 12  12
 14  14
 16  16
 18  18
 20  20
[torch.DoubleTensor of dimension 10x2]

> x
  2
  4
  6
  8
 10
 12
 14
 16
 18
 20
[torch.DoubleTensor of dimension 10x1]
```

[result] expandAs([result,] tensor)

이 함수는 self:expand(tensor:size())와 같습니다.

[Tensor] repeatTensor([result,] sizes)

`sizes`는 `torch.LongStorage` 한 개 또는 숫자 여러 개일 수 있습니다. 한 텐서의 반복은 `result`가 미리 제공되지 않는 이상 새 메모리를 할당합니다. 그 경우, `Tensor`의 메모리 크기는 바뀝니다. `sizes`는 각 차원에서 그 텐서가 반복될 횟수를 지정합니다.

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

```
 0.7160
 0.6514
 0.0704
 0.7856
 0.7452
[torch.DoubleTensor of dimension 5]

> torch.repeatTensor(x,3,2)
 0.7160  0.6514  0.0704  0.7856  0.7452  0.7160  0.6514  0.0704  0.7856  0.7452
 0.7160  0.6514  0.0704  0.7856  0.7452  0.7160  0.6514  0.0704  0.7856  0.7452
 0.7160  0.6514  0.0704  0.7856  0.7452  0.7160  0.6514  0.0704  0.7856  0.7452
[torch.DoubleTensor of dimension 3x10]

> torch.repeatTensor(x,3,2,1)
(1,.,.) =
  0.7160  0.6514  0.0704  0.7856  0.7452
  0.7160  0.6514  0.0704  0.7856  0.7452

(2,.,.) =
  0.7160  0.6514  0.0704  0.7856  0.7452
  0.7160  0.6514  0.0704  0.7856  0.7452

(3,.,.) =
  0.7160  0.6514  0.0704  0.7856  0.7452
  0.7160  0.6514  0.0704  0.7856  0.7452
[torch.DoubleTensor of dimension 3x2x5]
```

[Tensor] squeeze([dim])

그 텐서에 있는 모든 싱글턴 차원들을 없앱니다. 만약 `dim`이 주어지면, 텐서에서 그 특정 차원을 줄여서 없앱니다.

```
x=torch.rand(2,1,2,1,2)
> x
(1,1,1,.,.) =
  0.6020  0.8897

(2,1,1,.,.) =
  0.4713  0.2645

(1,1,2,.,.) =
  0.4441  0.9792

(2,1,2,.,.) =
  0.5467  0.8648
[torch.DoubleTensor of dimension 2x1x2x1x2]

> torch.squeeze(x)
(1,.,.) =
  0.6020  0.8897
```

```
  0.4441  0.9792

(2,.,.) =
  0.4713  0.2645
  0.5467  0.8648
[torch.DoubleTensor of dimension 2x2x2]

> torch.squeeze(x,2)
(1,1,.,.) =
  0.6020  0.8897

(2,1,.,.) =
  0.4713  0.2645

(1,2,.,.) =
  0.4441  0.9792

(2,2,.,.) =
  0.5467  0.8648
[torch.DoubleTensor of dimension 2x2x1x2]
```

텐서 뷰 다루기 (Manipulating the tensor view)

각각의 메소드들은 주어진 텐서의 스토리지를 보는 또다른 방식인 텐서 하나를 리턴합니다. 그러므로, 그 서브텐서의 메모리를 바꾸면, 처음 텐서에도 영향을 미칩니다. 반대 경우도 마찬가지입니다.

이 메소드들은 메모리 복사를 하지 않아 매우 빠릅니다.

[result] view([result,] tensor, sizes)

`tensor`와 관련된 스토리지의 다른 차원들을 가진 뷰를 만듭니다. 만약 `result`가 인자로 전달되지 않으면, 새 텐서 하나가 리턴됩니다. 만약 `result`가 인자로 전달되면, `result`의 스토리지는 `tensor`의 스토리지를 가리키기기 위해 만들어집니다.

`sizes`는 `torch.LongStorage` 한 개 또는 숫자 여러 개일 수 있습니다. 만약 그 차원 중 하나가 -1이면, 그 차원의 크기는 나머지 요소들로 추론됩니다.

```
x = torch.zeros(4)
> x:view(2,2)
 0 0
 0 0
[torch.DoubleTensor of dimension 2x2]
```

```
> x:view(2,-1)
 0 0
 0 0
[torch.DoubleTensor of dimension 2x2]

> x:view(torch.LongStorage{2,2})
 0 0
 0 0
[torch.DoubleTensor of dimension 2x2]

> x
 0
 0
 0
 0
[torch.DoubleTensor of dimension 4]
```

[result] viewAs([result,] tensor, template)

뷰 하나를 만듭니다. 그 뷰는 `tensor`에 관련된 그 스토리지의 템플릿(`template`)과 같은 차원을 가집니다. 만약 `result`가 인자로 전달되지 않으면, 새 텐서 하나가 리턴됩니다. 만약 `result`가 인자로 전달되면, `tensor`의 스토리지를 가리키는 `result`의 스토리지가 만들어집니다.

```
x = torch.zeros(4)
y = torch.Tensor(2,2)
> x:viewAs(y)
 0 0
 0 0
[torch.DoubleTensor of dimension 2x2]
```

[Tensor] transpose(dim1, dim2)

차원 `dim1`과 `dim2`가 바뀐 텐서 하나를 리턴합니다. 2차원 텐서에 대해서는 편의 메소드 t()도 쓸 수 있습니다.

```
x = torch.Tensor(3,4):zero()
x:select(2,3):fill(7) -- 2차원의 3번째 열을 7로 채움
> x
 0  0  7  0
 0  0  7  0
 0  0  7  0
[torch.DoubleTensor of dimension 3x4]

y = x:transpose(1,2) -- 차원 1과 2를 바꿈
```

```
> y
 0  0  0
 0  0  0
 7  7  7
 0  0  0
[torch.DoubleTensor of dimension 4x3]

y:select(2,3):fill(8) -- 3번째 열을 8로 채움
> y
 0  0  8
 0  0  8
 7  7  8
 0  0  8
[torch.DoubleTensor of dimension 4x3]

> x                  -- x의 내용 또한 바뀌었습니다.
 0  0  7  0
 0  0  7  0
 8  8  8  8
[torch.DoubleTensor of dimension 3x4]
```

[Tensor] t()

2차원 텐서를 위한 transpose()의 편의 메소드. 인자로 입력되는 텐서는 반드시 2차원이어야 합니다. 차원 1과 2를 바꿉니다.

```
x = torch.Tensor(3,4):zero()
x:select(2,3):fill(7)
y = x:t()
> y
 0  0  0
 0  0  0
 7  7  7
 0  0  0
[torch.DoubleTensor of dimension 4x3]

> x
 0  0  7  0
 0  0  7  0
 0  0  7  0
[torch.DoubleTensor of dimension 3x4]
```

[Tensor] permute(dim1, dim2, ..., dimn)

transpose()를 일반화합니다. permute()는 연속적 transpose() 호출을 대체하는 편의 메소드로 사용될 수 있습니다. permute()는 주어진 (dim1, dim2, ... , dimn)에 따라 차원들이 치환된(permuted) 텐서 하나를

리턴합니다. 그 순열은 반드시 완전히 지정되어야 합니다. 다시 말해, 그 텐서가 가진 차원의 개수와 파라미터 개수가 같아야 합니다.

```
x = torch.Tensor(3,4,2,5)
> x:size()
 3
 4
 2
 5
[torch.LongStorage of size 4]

y = x:permute(2,3,1,4) -- y = x:transpose(1,3):transpose(1,2)와 같음
> y:size()
 4
 2
 3
 5
[torch.LongStorage of size 4]
```

[Tensor] unfold(dim, size, step)

unfold()는 차원 `dim`에 있는 `size` 크기의 모든 슬라이스를 가진 텐서 하나를 리턴합니다. 두 슬라이스 사이 스텝은 `step`으로 주어집니다.

만약 `sizedim`이 차원 `dim`의 원래 크기이면, 리턴된 텐서에 있는 차원 `dim`의 크기는 `(sizedim - size) / step + 1`일 것입니다.

리턴된 텐서에는 차원 크기가 `size`가 되도록 추가적 차원이 덧붙여집니다.

```
x = torch.Tensor(7)
for i=1,7 do x[i] = i end
> x
 1
 2
 3
 4
 5
 6
 7
[torch.DoubleTensor of dimension 7]

> x:unfold(1, 2, 1)
 1  2
 2  3
 3  4
 4  5
 5  6
```

```
 6  7
[torch.DoubleTensor of dimension 6x2]

> x:unfold(1, 2, 2)
 1  2
 3  4
 5  6
[torch.DoubleTensor of dimension 3x2]
```

함수를 텐서에 적용하기 (Applying a function to a tensor)

이 함수들은 그 메소드를 호출한 텐서(self)의 각 요소에 함수를 적용합니다. 이 메소드들은 루아(`Lua`)에서 `for` 루프를 사용하는 것보다 훨씬 빠릅니다. (만약 함수가 무언가를 리턴하면) 결과는 `self`에 저장됩니다.

[self] apply(function)

self의 모든 요소에 인자로 입력된 함수를 적용합니다.

그 함수는 숫자 하나(그 텐서의 현재 요소)를 입력받아서, 숫자 하나를 리턴할 수도 있습니다. 그 경우, 그 리턴된 숫자는 self에 저장될 것입니다.

예:

```
i = 0
z = torch.Tensor(3,3)
z:apply(function(x)
  i = i + 1
  return i
end)              -- 텐서 z를 채움
> z
 1  2  3
 4  5  6
 7  8  9
[torch.DoubleTensor of dimension 3x3]

z:apply(math.sin) -- 사인 함수 적용
> z
 0.8415  0.9093  0.1411
-0.7568 -0.9589 -0.2794
 0.6570  0.9894  0.4121
[torch.DoubleTensor of dimension 3x3]

sum = 0
z:apply(function(x)
  sum = sum + x
```

```
end)              -- 요소들의 합 계산
> sum
1.9552094821074

> z:sum()         -- 실제로 정확합니다!
1.9552094821074
```

[self] map(tensor, function(xs, xt))

인자로 입력된 함수를 self와 `tensor`의 모든 요소에 적용합니다. 두 텐서의 요소 개수는 반드시 같아야 합니다. 그러나 두 텐서의 차원(sizes)은 다를 수도 있습니다.

함수는 숫자 두 개(self와 `tensor`의 현재 요소)를 입력받아서 숫자 하나를 리턴할 수도 있습니다. 그 경우, 그 리턴된 숫자는 self에 저장될 것입니다.

예:

```
x = torch.Tensor(3,3)
y = torch.Tensor(9)
i = 0
x:apply(function() i = i + 1; return i end) -- x 채움
i = 0
y:apply(function() i = i + 1; return i end) -- y 채움
> x
 1  2  3
 4  5  6
 7  8  9
[torch.DoubleTensor of dimension 3x3]

> y
 1
 2
 3
 4
 5
 6
 7
 8
 9
[torch.DoubleTensor of dimension 9]

x:map(y, function(xx, yy) return xx*yy end) -- 요소별 곱셈
> x
  1   4   9
 16  25  36
 49  64  81
[torch.DoubleTensor of dimension 3x3]
```

[self] map2(tensor1, tensor2, function(x, xt1, xt2))

인자로 입력된 함수를 self, `tensor1`, 그리고 `tensor2`의 모든 요소에 적용합니다. 모든 텐서의 요소 개수는 반드시 같아야 합니다. 그러나 차원 구성은 다를 수도 있습니다.

함수는 세 개의 숫자(self, `tensor1`, 그리고 `tensor2`의 현재 요소)를 입력받아서 숫자 하나를 리턴할 수도 있습니다. 그 경우, 그 리턴된 숫자는 self에 저장될 것입니다.

예:

```
x = torch.Tensor(3,3)
y = torch.Tensor(9)
z = torch.Tensor(3,3)

i = 0; x:apply(function() i = i + 1; return math.cos(i)*math.cos(i) end)
i = 0; y:apply(function() i = i + 1; return i end)
i = 0; z:apply(function() i = i + 1; return i end)

> x
 0.2919  0.1732  0.9801
 0.4272  0.0805  0.9219
 0.5684  0.0212  0.8302
[torch.DoubleTensor of dimension 3x3]

> y
 1
 2
 3
 4
 5
 6
 7
 8
 9
[torch.DoubleTensor of dimension 9]

> z
 1  2  3
 4  5  6
 7  8  9
[torch.DoubleTensor of dimension 3x3]

x:map2(y, z, function(xx, yy, zz) return xx+yy*zz end)
> x
  1.2919   4.1732   9.9801
 16.4272  25.0805  36.9219
 49.5684  64.0212  81.8302
[torch.DoubleTensor of dimension 3x3]
```

텐서 하나를 여러 텐서로 구성된 테이블로 나누기 (Dividing a tensor into a table of tensors)

이 함수들은 텐서 하나를 여러 개의 텐서들로 나눕니다. 그리고 그 나눠진 텐서들로 구성된 테이블 하나를 리턴합니다.

[result] split([result,] tensor, size, [dim])

텐서 `tensor`를 차원 `dim`을 따라 크기가 `size`(마지막 텐서의 경우 더 작을 수 있음)인 텐서들로 나눠 `result` 테이블 하나를 구성합니다. `dim`이 설정되지 않은 차원의 크기는 바뀌지 않습니다. 내부적으로, 차원 `dim`을 따라 연속적 narrow가 수행됩니다. 기본적으로, `dim`은 1로 설정됩니다.

만약 `result`가 인자로 전달되지 않으면, 새 테이블 하나가 리턴됩니다. 만약 `result`가 인자로 전달되면, 그 `result`는 비워져 재사용됩니다.

예:

```
x = torch.randn(3,4,5)

> x:split(2,1)
{
  1 : DoubleTensor - size: 2x4x5
  2 : DoubleTensor - size: 1x4x5
}

> x:split(3,2)
{
  1 : DoubleTensor - size: 3x3x5
  2 : DoubleTensor - size: 3x1x5
}

> x:split(2,3)
{
  1 : DoubleTensor - size: 3x4x2
  2 : DoubleTensor - size: 3x4x2
  3 : DoubleTensor - size: 3x4x1
}
```

[result] chunk([result,] tensor, n, [dim])

텐서 `tensor`를 차원 `dim`을 따라 대략 같은 크기의 덩어리(chunk) `n` 개로 나눕니다. 그리고 그 덩어리들을 텐서들로 구성된 `result` 테이블 한 개로 리턴합니다. 기본적으로, 인자 dim은 1로 설정됩니다.

내부적으로, 이 함수는 스플릿(split)을 사용합니다.

```
torch.split(result, tensor, math.ceil(tensor:size(dim)/n), dim)
```

예:

```
x = torch.randn(3,4,5)

> x:chunk(2,1)
{
  1 : DoubleTensor - size: 2x4x5
  2 : DoubleTensor - size: 1x4x5
}

> x:chunk(2,2)
{
  1 : DoubleTensor - size: 3x2x5
  2 : DoubleTensor - size: 3x2x5
}

> x:chunk(2,3)
{
  1 : DoubleTensor - size: 3x4x3
  2 : DoubleTensor - size: 3x4x2
}
```

LuaJIT FFI 접근 (LuaJIT FFI access)

이 함수들은 토치의 텐서와 스토리지 자료 구조를 LuaJIT FFI를 통해 드러냅니다. LuaJIT FFI는 모든 루아에서 텐서와 스토리지로의 극도로 빠른 접근을 가능하게 합니다.

[result] data(tensor, [asnumber])

`tensor`의 가공되지 않은(raw) 데이터를 가리키는 LuaJIT FFI 포인터 하나를 리턴합니다. 만약 `asnumber`가 true이면, `intptr_t` cdata를 리턴합니다. 우리는 그것을 `tonumber()`로 루아 수(Lua number)로 바꿀 수 있습니다.

텐서의 가공되지 않은 데이터에 이렇게 접근하는 것은 극도로 효율적입니다. 많은 경우에 이 접근 방식은 거의 C만큼 빠릅니다.

예:

```
t = torch.randn(3,2)
> t
 0.8008 -0.6103
 0.6473 -0.1870
-0.0023 -0.4902
[torch.DoubleTensor of dimension 3x2]

t_data = torch.data(t)
for i = 0,t:nElement()-1 do t_data[i] = 0 end
> t
 0 0
 0 0
 0 0
[torch.DoubleTensor of dimension 3x2]
```

주의: 명심하십시오. 이렇게 가공되지 않은 데이터에 접근하는 것은 위험합니다. 그리고 이는 오직 연속적인 텐서들에만 적용되어야 합니다. 만약 한 텐서가 연속적이지 않으면, 당신은 반드시 그 텐서의 크기와 스트라이드 정보를 사용해야 합니다. 한 텐서가 연속적임을 확인하는 법은 쉽습니다.

```
t = torch.randn(3,2)
t_noncontiguous = t:transpose(1,2)

-- torch.data(t_noncontiguous)로 이것을 하는 것은 안전하지 않을 수 있습니다.
t_transposed_and_contiguous = t_noncontiguous:contiguous()

-- 이제 가공되지 않은(raw) 포인터로 작업해도 안전합니다
data = torch.data(t_transposed_and_contiguous)
```

마지막으로, 그 포인터는 평범한 `intptr_t` cdata 하나로 리턴될 수 있습니다. 이것은 스레드들 사이에 포인터들을 공유하는 데 유용할 수 있습니다. (주의: 이것은 위험합니다, 왜냐하면 두 번째 텐서가 그 스토리지에 대한 참조 카운터를 증가시키지 않기 때문입니다. 만약 첫 번째 텐서가 free되면, 두 번째 텐서의 데이터는 허상 포인터가 됩니다).

```
t = torch.randn(10)
p = tonumber(torch.data(t,true))
s = torch.Storage(10, p)
tt = torch.Tensor(s)
-- tt와 t는 같은 데이터에 대한 뷰(view) 하나입니다.
```

[result] cdata(tensor, [asnumber])

`tensor`의 C 구조를 가리키는 LuaJIT FFI 포인터를 리턴합니다. 이것을 조심히 사용하십시오. 그리고 FFI.lua에서 `tensor`의 멤버들을 자세히 보십시오.

참조된 횟수 세기 (Reference counting)

루아는 텐서들의 참조 횟수를 셉니다. 한 객체(C 또는 루아 상태)가 한 텐서를 참조하여 보유할 필요가 있을 때마다, 그에 상응하는 텐서 참조 카운터가 증가됩니다. 그 참조 카운터는 그 텐서가 더 이상 필요 없어질 때 감소됩니다.

이 메소드들은 각별히 조심해서 사용해야 합니다. 일반적으로, 당신이 의도적으로 사용하지 않는 이상, 이 메소드들은 결코 쓰일 일이 없습니다. 참조들은 자동으로 조작되기 때문입니다. 이 메소드들은 스레드 환경에서 유용할 수 있습니다. 이 메소드들은 atomic 연산임을 유념하십시오.

retain()

그 텐서의 참조 카운터를 증가.

free()

그 텐서의 참조 카운터를 감소. 만약 카운터가 0이면, 그 텐서를 free(그 텐서에 할당된 메모리를 회수)함.

목차

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