CmdLine

This class provides a parameter parsing framework which is very useful when one needs to run several experiments that rely on different parameter settings that are passed in the command line. This class will also override the default print function to direct all the output to a log file as well as screen at the same time.

A sample `lua` file is given below that makes use of `CmdLine` class.

```
cmd = torch.CmdLine()
cmd:text()
cmd:text()
cmd:text('Training a simple network')
cmd:text()
cmd:text('Options')
cmd:option('-seed',123,'initial random seed')
cmd:option('-booloption',false,'boolean option')
cmd:option('-stroption','mystring','string option')
cmd:text()

-- parse input params
params = cmd:parse(arg)

params.rundir = cmd:string('experiment', params, {dir=true})
paths.mkdir(params.rundir)

-- create log file
cmd:log(params.rundir .. '/log', params)
```

When this file is run on the th command line as follows

```
# th myscript.lua
```

It will produce the following output:

```
[program started on Tue Jan 10 15:33:49 2012]
```

```
[command line arguments]
booloption  false
seed    123
rundir  experiment
stroption   mystring
[----------------------]
booloption  false
seed    123
rundir  experiment
stroption   mystring
```

The same output will also be written to file `experiment/log`. Whenever one of the options are passed on the command line and is different than the default value, the `rundir` is name is produced to reflect the parameter setting.

```
# th myscript.lua -seed 456 -stroption mycustomstring
```

This will produce the following output:

```
[program started on Tue Jan 10 15:36:55 2012]
[command line arguments]
booloption  false
seed    456
rundir  experiment,seed=456,stroption=mycustomstring
stroption   mycustomstring
[----------------------]
booloption  false
seed    456
rundir  experiment,seed=456,stroption=mycustomstring
stroption   mycustomstring
```

and the output will be logged in `experiment,seed=456,stroption=mycustomstring/log`

addTime([name] [,format])

Adds a prefix to every line in the log file with the date/time in the given format with an optional name argument. The date/time format is the same as `os.date()`. Note that the prefix is only added to the log file, not the screen output. The default value for name is empty and the default format is '%F %T'.

The final produced output for the following command is:

```
> cmd:addTime('your project name','%F %T')
> print('Your log message')
```

```
2012-02-07 08:21:56[your project name]: Your log message
```

log(filename, parameter_table)

It sets the log filename to `filename` and prints the values of parameters in the `parameter_table`. If filename is an open file descriptor, it will write to the file instead of creating a new one.

option(name, default, help)

Stores an option argument. The name should always start with '-'.

[table] parse(arg)

Parses a given table, `arg` is by default the argument table that is created by `lua` using the command line arguments passed to the executable. Returns a table of option values.

silent()

Silences the output to standard output. The only output is written to the log file.

[string] string(prefix, params, ignore)

Returns a string representation of the options by concatenating the non-default options. `ignore` is a table `{dir=true}`, which will ensure that option named `dir` will be ignored while creating the string representation.

This function is useful for creating unique experiment directories that depend on the parameter settings.

text(string)

Logs a custom text message.

Directory Functions

The following functions can be used
to examine directory contents or manipulate directories.

paths.dir(dname)

Return a table containing the files and directories in directory `dname`.
This function return `nil` if the specified directory does not exists.
For linux, this includes the `.` and `..` directories.

paths.files(dname [, include])

Returns an iterator over the files and directories located in directory `dname`.
For linux, this includes the `.` and `..` directories.
This can be used in ***for*** expression as shown below:

```
for f in paths.files(".") do
   print(f)
end
```

Optional argument `include` is either a function or a string used to
determine which files are to be included. The function takes the filename
as argument and should return true if the file is to be included.
When a string is provided, the following function is used :

```
function(file)
   return file:find(f)
end
```

Files and directories of sub-folders aren't included.

paths.iterdirs(dname)

Returns an iterator over the directories located in directory `dname`.
This can be used in ***for*** expression as shown below:

```
for dir in paths.iterdirs(".") do
   print(dir)
end
```

Directories of sub-folders, and the `.` and `..` folders aren't included.

paths.iterfiles(dname)

Returns an iterator over the files (non-directories) located in directory `dname`.
This can be used in ***for*** expression as shown below:

```
for file in paths.iterfiles(".") do
   print(file)
end
```

Files of sub-folders, and the `.` and `..` folders aren't included.

paths.mkdir(s)

Create a directory.
Returns `true` on success.

paths.rmdir(s)

Delete an empty directory.
Returns `true` on success.

paths.rmall(s, y)

Recursively delete file or directory `s` and its contents.

Argument `y` must be string `"yes"`
Returns `true` on success.

Math Functions

Torch provides MATLAB-like functions for manipulating `Tensor` objects. Functions fall into several types of categories:

- Constructors like `zeros`, `ones`;
- Extractors like `diag` and `triu`;
- Element-wise mathematical operations like `abs` and `pow`;
- BLAS operations;
- Column or row-wise operations like `sum` and `max`;
- Matrix-wide operations like `trace` and `norm`;
- Convolution and cross-correlation operations like `conv2`;
- Basic linear algebra operations like `eig`;
- Logical operations on `Tensor` s.

By default, all operations allocate a new `Tensor` to return the result.
However, all functions also support passing the target `Tensor` (s) as the first argument(s), in which case the target `Tensor` (s) will be resized accordingly and filled with result.
This property is especially useful when one wants have tight control over when memory is allocated.

The *Torch* package adopts the same concept, so that calling a function directly on the `Tensor` itself using an object-oriented syntax is equivalent to passing the `Tensor` as the optional resulting `Tensor`.
The following two calls are equivalent.

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

Similarly, `torch.conv2` function can be used in the following manner.

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

> res2 = torch.Tensor()
> torch.conv2(res2, x, k)     -- case 2

> res2:dist(res1)
0
```

The advantage of second case is, same `res2` `Tensor` can be used successively in a loop without any new allocation.

```
-- no new memory allocations...
> for i = 1, 100 do
     torch.conv2(res2, x, k)
  end

> res2:dist(res1)
0
```

Construction or extraction functions

[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])` returns a `Tensor` `x` which is the concatenation of `Tensor` s `x_1` and `x_2` along dimension `dimension`.

If `dimension` is not specified or if it is `-1`, it is the maximum last dimension over all input tensors, except if all tensors are empty, then it is `1`.

The other dimensions of `x_1` and `x_2` have to be equal.

Also supports arrays with arbitrary numbers of `Tensor` s as inputs.

Empty tensors are ignored during catting, and thus do not throw an error. Performing cat on empty tensors only will always result in an empty tensor.

Examples:

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

```
 1
 0
 0
[torch.DoubleTensor of size 5]

> torch.cat(torch.ones(3, 2), torch.zeros(2, 2), 1)
 1  1
 1  1
 1  1
 0  0
 0  0
[torch.DoubleTensor of size 5x2]

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

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

> 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 size 7x2]

> 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 size 7x2]

> torch.cat({torch.Tensor(), torch.rand(3, 2)}, 1)
 0.3227  0.0493
 0.9161  0.1086
 0.2206  0.7449
[torch.DoubleTensor of size 3x2]
```

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

`y = torch.diag(x)` when `x` is of dimension 1 returns a diagonal matrix with diagonal elements constructed from `x`.

`y = torch.diag(x)` when `x` is of dimension 2 returns a `Tensor` of dimension 1 with elements constructed from the diagonal of `x`.

`y = torch.diag(x, k)` returns the k-th diagonal of `x`, where `k = 0` is the main diagonal, `k > 0` is above the main diagonal and `k < 0` is below the main diagonal.

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

`y = torch.eye(n)` returns the `n × n` identity matrix.

`y = torch.eye(n, m)` returns an `n × m` identity matrix with ones on the diagonal and zeros elsewhere.

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

`y = torch.histc(x)` returns the histogram of the elements in `x`.
By default the elements are sorted into 100 equally spaced bins between the minimum and maximum values of `x`.

`y = torch.histc(x, n)` same as above with `n` bins.

`y = torch.histc(x, n, min, max)` same as above with `n` bins and `[min, max]` as elements range.

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

`y = torch.bhistc(x)` returns the histogram of the elements in 2d tensor `x` along the last dimension.
By default the elements are sorted into 100 equally spaced bins between the minimum and maximum values of `x`.

`y = torch.bhistc(x, n)` same as above with `n` bins.

`y = torch.bhistc(x, n, min, max)` same as above with `n` bins and `[min, max]` as elements range.

```
x =torch.Tensor(3, 6)

> x[1] = torch.Tensor{ 2, 4, 2, 2, 5, 4 }
> x[2] = torch.Tensor{ 3, 5, 1, 5, 3, 5 }
> x[3] = torch.Tensor{ 3, 4, 2, 5, 5, 1 }

> x
 2  4  2  2  5  4
 3  5  1  5  3  5
 3  4  2  5  5  1
[torch.DoubleTensor of size 3x6]

> torch.bhistc(x, 5, 1, 5)
 0  3  0  2  1
 1  0  2  0  3
 1  1  1  1  2
[torch.DoubleTensor of size 3x5]

> y = torch.Tensor(1, 6):copy(x[1])

> torch.bhistc(y, 5)
 3  0  2  0  1
[torch.DoubleTensor of size 1x5]
```

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

`y = torch.linspace(x1, x2)` returns a one-dimensional `Tensor` of size 100 equally spaced points between `x1` and `x2`.

`y = torch.linspace(x1, x2, n)` returns a one-dimensional `Tensor` of `n` equally spaced points between `x1` and `x2`.

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

`y = torch.logspace(x1, x2)` returns a one-dimensional `Tensor` of `100` logarithmically eqally spaced points between `10^x1` and `10^x2`.

`y = torch.logspace(x1, x2, n)` returns a one-dimensional `Tensor` of `n` logarithmically equally spaced points between `10^x1` and `10^x2`.

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

`y = torch.multinomial(p, n)` returns a `Tensor` `y` where each row contains `n` indices sampled from the multinomial probability distribution located in the corresponding row of `Tensor` `p`.

The rows of `p` do not need to sum to one (in which case we use the values as weights), but must be non-negative and have a non-zero sum.
Indices are ordered from left to right according to when each was sampled (first samples are placed in first column).

If `p` is a vector, `y` is a vector size `n`.

If `p` is a m-rows matrix, `y` is an `m × n` matrix.

If `replacement` is `true`, samples are drawn **with replacement**.
If not, they are drawn **without replacement**, which means that when a sample index is drawn for a row, it cannot be drawn again for that row.
This implies the constraint that `n` must be lower than `p` length (or number of columns of `p` if it is a matrix).

The default value for `replacement` is `false`.

```
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
```

Note: If you use the function with a given result `Tensor`, i.e. of the function prototype: `torch.multinomial(res, p, n [, replacement])` then you will have to call it slightly differently as:

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

This is due to the fact that the result here is of a `LongTensor` type, and we do not define a `torch.multinomial` over long `Tensor` s.

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

`y = torch.ones(n)` returns a one-dimensional `Tensor` of size `n` filled with ones.

`y = torch.ones(m, n)` returns a `m × n` `Tensor` filled with ones.

For more than `4` dimensions, you can use a storage as argument: `y = torch.ones(torch.LongStorage{m, n, k, l, o})`.

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

`y = torch.rand(n)` returns a one-dimensional `Tensor` of size `n` filled with random numbers from a uniform distribution on the interval `[0, 1)`.

`y = torch.rand(m, n)` returns a `m × n` `Tensor` of random numbers from a uniform distribution on the interval `[0, 1)`.

For more than 4 dimensions, you can use a storage as argument: `y = torch.rand(torch.LongStorage{m, n, k, l, o})`.

`y = torch.rand(gen, m, n)` returns a `m × n` `Tensor` of random numbers from a uniform distribution on the interval `[0, 1)`, using a non-global random number generator `gen` created by torch.Generator().

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

`y = torch.randn(n)` returns a one-dimensional `Tensor` of size `n` filled with random numbers from a normal distribution with mean zero and variance one.

`y = torch.randn(m, n)` returns a `m × n` `Tensor` of random numbers from a normal distribution with mean zero and variance one.

For more than 4 dimensions, you can use a storage as argument: `y = torch.randn(torch.LongStorage{m, n, k, l, o})`.

`y = torch.randn(gen, m, n)` returns a `m × n` `Tensor` of random numbers from a normal distribution with mean zero and variance one, using a non-global random number generator `gen` created by torch.Generator().

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

`y = torch.range(x, y)` returns a `Tensor` of size `floor((y - x) / step) + 1` with values from `x` to `y` with step `step` (default to 1).

```
> torch.range(2, 5)
 2
 3
 4
 5
[torch.DoubleTensor of size 4]

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

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

`y = torch.randperm(n)` returns a random permutation of integers from 1 to `n`.

`y = torch.randperm(gen, n)` returns a random permutation of integers from 1 to `n`, using a non-global random number generator `gen` created by torch.Generator().

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

`y = torch.reshape(x, m, n)` returns a new `m × n` `Tensor` y whose elements are taken rowwise from `x`, which must have `m * n` elements. The elements are copied into the new `Tensor`.

For more than 4 dimensions, you can use a storage: `y = torch.reshape(x, torch.LongStorage{m, n, k, l, o})`.

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

`y = torch.tril(x)` returns the lower triangular part of `x`, the other elements of `y` are set to 0.

`torch.tril(x, k)` returns the elements on and below the k-th diagonal of `x` as non-zero. `k = 0` is the main diagonal, `k > 0` is above the main diagonal and `k < 0` is below the main diagonal.

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

`y = torch.triu(x)` returns the upper triangular part of `x`, the other elements of `y` are set to 0.

`torch.triu(x, k)` returns the elements on and above the k-th diagonal of `x` as non-zero. `k = 0` is the main diagonal, `k > 0` is above the main diagonal and `k < 0` is below the main diagonal.

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

`y = torch.zeros(n)` returns a one-dimensional `Tensor` of size n filled with zeros.

`y = torch.zeros(m, n)` returns a `m × n` `Tensor` filled with zeros.

For more than 4 dimensions, you can use a storage: `y = torch.zeros(torch.LongStorage{m, n, k, l, o})`.

Element-wise Mathematical Operations

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

`y = torch.abs(x)` returns a new `Tensor` with the absolute values of the elements of `x`.

`x:abs()` replaces all elements in-place with the absolute values of the elements of `x`.

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

`y = torch.sign(x)` returns a new `Tensor` with the sign (`+/- 1`) of the elements of `x`.

`x:sign()` replaces all elements in-place with the sign of the elements of `x`.

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

`y = torch.acos(x)` returns a new `Tensor` with the arcosine of the elements of `x`.

`x:acos()` replaces all elements in-place with the arcosine of the elements of `x`.

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

`y = torch.asin(x)` returns a new `Tensor` with the arcsine of the elements of `x`.

`x:asin()` replaces all elements in-place with the arcsine of the elements of `x`.

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

`y = torch.atan(x)` returns a new `Tensor` with the arctangent of the elements of `x`.

`x:atan()` replaces all elements in-place with the arctangent of the elements of `x`.

[res] torch.atan2([res,] x, y)

`y = torch.atan2(x, y)` returns a new `Tensor` with the arctangent of the elements of `x` and `y`.

`x:atan2()` replaces all elements in-place with the arctangent of the elements of `x` and `y`.

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

`y = torch.ceil(x)` returns a new `Tensor` with the values of the elements of `x` rounded up to the nearest integers.

`x:ceil()` replaces all elements in-place with the values of the elements of `x` rounded up to the nearest integers.

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

`y = torch.cos(x)` returns a new `Tensor` with the cosine of the elements of `x`.

`x:cos()` replaces all elements in-place with the cosine of the elements of `x`.

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

`y = torch.cosh(x)` returns a new `Tensor` with the hyberbolic cosine of the elements of `x`.

`x:cosh()` replaces all elements in-place with the hyberbolic cosine of the elements of `x`.

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

`y = torch.exp(x)` returns, for each element in `x`, *e* (*Neper number*, the base of natural logarithms) raised to the power of the element in `x`.

`x:exp()` returns, for each element in `x`, *e* raised to the power of the element in `x`.

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

`y = torch.floor(x)` returns a new `Tensor` with the values of the elements of `x` rounded down to the nearest integers.

`x:floor()` replaces all elements in-place with the values of the elements of `x` rounded down to the nearest integers.

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

`y = torch.log(x)` returns a new `Tensor` with the natural logarithm of the elements of `x`.

`x:log()` replaces all elements in-place with the natural logarithm of the elements of `x`.

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

`y = torch.log1p(x)` returns a new `Tensor` with the natural logarithm of the elements of `x + 1`.

`x:log1p()` replaces all elements in-place with the natural logarithm of the elements of `x + 1`.
This function is more accurate than `log` for small values of `x`.

x:neg()

`x:neg()` replaces all elements in-place with the sign-reversed values of the elements of `x`.

x:cinv()

`x:cinv()` replaces all elements in-place with `1.0 / x`.

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

Let `x` be a `Tensor` and `n` a number.

`y = torch.pow(x, n)` returns a new `Tensor` with the elements of `x` to the power of `n`.

`y = torch.pow(n, x)` returns, a new `Tensor` with `n` to the power of the elements of `x`.

`x:pow(n)` replaces all elements in-place with the elements of `x` to the power of `n`.

`torch.pow(x, n, x)` replaces all elements in-place with `n` to the power of the elements of `x`.

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

`y = torch.round(x)` returns a new `Tensor` with the values of the elements of `x` rounded to the nearest integers.

`x:round()` replaces all elements in-place with the values of the elements of `x` rounded to the nearest integers.

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

`y = torch.sin(x)` returns a new `Tensor` with the sine of the elements of `x`.

`x:sin()` replaces all elements in-place with the sine of the elements of `x`.

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

`y = torch.sinh(x)` returns a new `Tensor` with the hyperbolic sine of the elements of `x`.

`x:sinh()` replaces all elements in-place with the hyperbolic sine of the elements of `x`.

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

`y = torch.sqrt(x)` returns a new `Tensor` with the square root of the elements of `x`.

`x:sqrt()` replaces all elements in-place with the square root of the elements of `x`.

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

`y = torch.rsqrt(x)` returns a new `Tensor` with the reciprocal of the square root of the elements of `x`.

`x:rsqrt()` replaces all elements in-place with the reciprocal of the square root of the elements of `x`.

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

`y = torch.tan(x)` returns a new `Tensor` with the tangent of the elements of `x`.

`x:tan()` replaces all elements in-place with the tangent of the elements of `x`.

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

`y = torch.tanh(x)` returns a new `Tensor` with the hyperbolic tangent of the elements of `x`.

`x:tanh()` replaces all elements in-place with the hyperbolic tangent of the elements of `x`.

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

`y = torch.sigmoid(x)` returns a new `Tensor` with the sigmoid of the elements of `x`.

`x:sigmoid()` replaces all elements in-place with the sigmoid of the elements of `x`.

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

`y = torch.trunc(x)` returns a new `Tensor` with the truncated integer values of the elements of `x`.

`x:trunc()` replaces all elements in-place with the truncated integer values of the elements of `x`.

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

`y = torch.frac(x)` returns a new `Tensor` with the fractional portion of the elements of `x`.

`x:frac()` replaces all elements in-place with the fractional portion of the elements of `x`.

Basic operations

In this section, we explain basic mathematical operations for `Tensor`s.

[boolean] equal([tensor1,] tensor2)

Returns `true` iff the dimensions and values of `tensor1` and `tensor2` are exactly the same.

```
x = torch.Tensor{1,2,3}
y = torch.Tensor{1,2,3}
> x:equal(y)
true

y = torch.Tensor{1,2,4}
> x:equal(y)
false
```

Note that `a:equal(b)` is more efficient that `a:eq(b):all()` as it avoids allocation of a temporary tensor and can short-circuit.

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

Add the given value to all elements in the `Tensor`.

`y = torch.add(x, value)` returns a new `Tensor`.

`x:add(value)` add `value` to all elements in place.

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

Add `tensor1` to `tensor2` and put result into `res`.
The number of elements must match, but sizes do not matter.

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

`y = torch.add(a, b)` returns a new `Tensor`.

`torch.add(y, a, b)` puts `a + b` in `y`.

`a:add(b)` accumulates all elements of `b` into `a`.

`y:add(a, b)` puts `a + b` in `y`.

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

Multiply elements of `tensor2` by the scalar `value` and add it to `tensor1`.
The number of elements must match, but sizes do not matter.

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

`x:add(value, y)` multiply-accumulates values of `y` into `x`.

`z:add(x, value, y)` puts the result of `x + value * y` in `z`.

`torch.add(x, value, y)` returns a new `Tensor` `x + value * y`.

`torch.add(z, x, value, y)` puts the result of `x + value * y` in `z`.

tensor:csub(value)

Subtracts the given value from all elements in the `Tensor`, in place.

tensor:csub(tensor2)

Subtracts `tensor2` from `tensor`, in place.
The number of elements must match, but sizes do not matter.

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

`a:csub(b)` put `a - b` into `a`.

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

Multiply all elements in the `Tensor` by the given `value`.

`z = torch.mul(x, 2)` will return a new `Tensor` with the result of `x * 2`.

`torch.mul(z, x, 2)` will put the result of `x * 2` in `z`.

`x:mul(2)` will multiply all elements of `x` with `2` in-place.

`z:mul(x, 2)` will put the result of `x * 2` in `z`.

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

Clamp all elements in the `Tensor` into the range `[min_value, max_value]`. ie:

```
        ⎧ min_value, if x_i < min_value
y_i =   ⎨ x_i,       if min_value ≤ x_i ≤ max_value
        ⎩ max_value, if x_i > max_value
```

`z = torch.clamp(x, 0, 1)` will return a new `Tensor` with the result of `x` bounded between `0` and `1`.

`torch.clamp(z, x, 0, 1)` will put the result in `z`.

`x:clamp(0, 1)` will perform the clamp operation in place (putting the result in `x`).

`z:clamp(x, 0, 1)` will put the result in `z`.

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

Element-wise multiplication of `tensor1` by `tensor2`.
The number of elements must match, but sizes do not matter.

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

`z = torch.cmul(x, y)` returns a new `Tensor`.

`torch.cmul(z, x, y)` puts the result in `z`.

`y:cmul(x)` multiplies all elements of `y` with corresponding elements of `x`.

`z:cmul(x, y)` puts the result in `z`.

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

Element-wise power operation, taking the elements of `tensor1` to the powers given by elements of `tensor2`.
The number of elements must match, but sizes do not matter.

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

`z = torch.cpow(x, y)` returns a new `Tensor`.

`torch.cpow(z, x, y)` puts the result in `z`.

`y:cpow(x)` takes all elements of `y` to the powers given by the corresponding elements of `x`.

`z:cpow(x, y)` puts the result in `z`.

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

Performs the element-wise multiplication of `tensor1` by `tensor2`, multiply the result by the scalar `value` (1 if not present) and add it to `x`.
The number of elements must match, but sizes do not matter.

```
> 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.DoubleTensor of size 2x2]
```

`z:addcmul(value, x, y)` accumulates the result in `z`.

`torch.addcmul(z, value, x, y)` returns a new `Tensor` with the result.

`torch.addcmul(z, z, value, x, y)` puts the result in `z`.

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

Divide all elements in the `Tensor` by the given `value`.

`z = torch.div(x, 2)` will return a new `Tensor` with the result of `x / 2`.

`torch.div(z, x, 2)` will put the result of `x / 2` in `z`.

`x:div(2)` will divide all elements of `x` with `2` in-place.

`z:div(x, 2)` puts the result of `x / 2` in `z`.

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

Performs the element-wise division of `tensor1` by `tensor2`.
The number of elements must match, but sizes do not matter.

```
> 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.DoubleTensor of size 2x2]
```

`z = torch.cdiv(x, y)` returns a new `Tensor`.

`torch.cdiv(z, x, y)` puts the result in `z`.

`y:cdiv(x)` divides all elements of `y` with corresponding elements of `x`.

`z:cdiv(x, y)` puts the result in `z`.

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

Performs the element-wise division of `tensor1` by `tensor2`, multiply the result by the scalar `value` and add it to `x`.
The number of elements must match, but sizes do not matter.

```
> 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.DoubleTensor of size 2x2]
```

`z:addcdiv(value, x, y)` accumulates the result in `z`.

`torch.addcdiv(z, value, x, y)` returns a new `Tensor` with the result.

`torch.addcdiv(z, z, value, x, y)` puts the result in `z`.

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

Computes remainder of division (rounded towards zero) of all elements in the `Tensor` by `value`.
This works both for integer and floating point numbers. It behaves the same as Lua bulit-in function `math.fmod()` and a little bit different from `torch.remainder()` and `%` operator. For example:

```
> x = torch.Tensor({-3, 3})
> torch.fmod(x, 2)
-1
 1
[torch.DoubleTensor of size 2]

> torch.fmod(x, -2)
-1
 1
[torch.DoubleTensor of size 2]

> torch.remainder(x, 2)
 1
 1
```

```
[torch.DoubleTensor of size 2]

> torch.remainder(x, -2)
-1
-1
[torch.DoubleTensor of size 2]
```

`z = torch.fmod(x, 2)` will return a new `Tensor` with the result of `math.fmod(x, 2)`.

`torch.fmod(z, x, 2)` will put the result of `math.fmod(x, 2)` in `z`.

`x:fmod(2)` will replace all elements of `x` the result of `math.fmod(x, 2)` in-place.

`z:fmod(x, 2)` puts the result of `math.fmod(x, 2)` in `z`.

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

Computes remainder of division (rounded to nearest) of all elements in the `Tensor` by `value`. This works both for integer and floating point numbers. It behaves the same as `%` operator and can be expressed as `a % b = a - b * floor(a/b)`. See `torch.fmod()` for comparison.

`z = torch.remainder(x, 2)` will return a new `Tensor` with the result of `x % 2`.

`torch.remainder(z, x, 2)` will put the result of `x % 2` in `z`.

`x:remainder(2)` will replace all elements of `x` the result of `x % 2` in-place.

`z:remainder(x, 2)` puts the result of `x % 2` in `z`.

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

This function is deprecated and exists only for compatibility with previous versions. Please use `torch.fmod()` or `torch.remainder()` instead.

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

Computes the element-wise remainder of the division (rounded towards zero) of `tensor1` by `tensor2`.
The number of elements must match, but sizes do not matter.

```
> x = torch.Tensor({{3, 3}, {-3, -3}})
> y = torch.Tensor({{2, -2}, {2, -2}})
> x:cfmod(y)
 1  1
-1 -1
[torch.DoubleTensor of size 2x2]
```

`z = torch.cfmod(x, y)` returns a new `Tensor`.

`torch.cfmod(z, x, y)` puts the result in `z`.

`y:cfmod(x)` replaces all elements of `y` by their remainders of division (rounded towards zero) by
corresponding elements of `x`.

`z:cfmod(x, y)` puts the result in `z`.

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

Computes element-wise remainder of the division (rounded to nearest) of `tensor1` by `tensor2`.
The number of elements must match, but sizes do not matter.

```
> x = torch.Tensor({{3, 3}, {-3, -3}})
> y = torch.Tensor({{2, -2}, {2, -2}})
> x:cfmod(y)
 1  1
-1 -1
[torch.DoubleTensor of size 2x2]
```

`z = torch.cremainder(x, y)` returns a new `Tensor`.

`torch.cremainder(z, x, y)` puts the result in `z`.

`y:cremainder(x)` replaces all elements of `y` by their remainders of division (rounded to nearest) by
corresponding elements of `x`.

`z:cremainder(x, y)` puts the result in `z`.

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

This function is deprecated and exists only for compatibility with previous versions. Please use `torch.cfmod()` or `torch.cremainder()` instead.

[number] torch.dot(tensor1, tensor2)

Performs the dot product between `tensor1` and `tensor2`.
The number of elements must match: both `Tensor` s are seen as a 1D vector.

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

`torch.dot(x, y)` returns dot product of `x` and `y`.
`x:dot(y)` returns dot product of `x` and `y`.

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

Performs a matrix-vector multiplication between `mat` (2D `Tensor`) and `vec2` (1D `Tensor`) and add it to `vec1`.

Optional values `v1` and `v2` are scalars that multiply `vec1` and `vec2` respectively.

In other words,

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

Sizes must respect the matrix-multiplication operation: if `mat` is a `n × m` matrix, `vec2` must be vector of size `m` and `vec1` must be a vector of size `n`.

```
> 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.DoubleTensor of size 3]
```

`torch.addmv(x, y, z)` returns a new `Tensor` with the result.

`torch.addmv(r, x, y, z)` puts the result in `r`.

Differences when used as a method

`x:addmv(y, z)` does `x = x + y * z`

`r:addmv(x, y, z)` does `r = x + y * z` if x is a vector

`r:addmv(s, y, z)` does `r = r + s * y * z` if `s` is a scalar.

`r:addmv(x, s, y, z)` does `r = x + s * y * z` if `s` is a scalar and `x` is a vector.

`r:addmv(s1, s2, y, z)` does `r = s1 * r + s2 * y * z` if `s1` and `s2` are scalars.

The last example does not accurately fit into the function signature, and needs a special mention. It changes the function signature to:

`[vec1] = vec1:addmv([v1,] [v2,] mat, vec2)`

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

Performs the outer-product between `vec1` (1D `Tensor`) and `vec2` (1D `Tensor`).

Optional values `v1` and `v2` are scalars that multiply `mat` and `vec1 [out] vec2` respectively.

In other words,

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

If `vec1` is a vector of size `n` and `vec2` is a vector of size `m`, then `mat` must be a matrix of size `n × m`.

```
> x = torch.range(1, 3)
```

```
> y = torch.range(1, 2)
> M = torch.Tensor(3, 2):zero()
> M:addr(x, y)
 1  2          --     |0 0|     |1 2|
 2  4          -- = 1*|0 0| + 1*|2 4|
 3  6          --     |0 0|     |3 6|
[torch.DoubleTensor of size 3x2]
-- default values of v1 and v2 are 1.

> M:addr(2, 1, x, y)
  3   6         --     |1 2|     |1 2|
  6  12         -- = 2*|2 4| + 1*|2 4|
  9  18         --     |3 6|     |3 6|
[torch.DoubleTensor of size 3x2]

> A = torch.range(1, 6):resize(3, 2)
> A
 1  2
 3  4
 5  6
[torch.DoubleTensor of size 3x2]
> M:addr(2, A, 1, x, y)
  3   6         --   |1 2|     |1 2|
  8  12         -- 2*|3 4| + 1*|2 4|
 13  18         --   |5 6|     |3 6|
[torch.DoubleTensor of size 3x2]
```

`torch.addr(M, x, y)` returns the result in a new `Tensor`.

`torch.addr(r, M, x, y)` puts the result in `r`.

`M:addr(x, y)` puts the result in `M`.

`r:addr(M, x, y)` puts the result in `r`.

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

Performs a matrix-matrix multiplication between `mat1` (2D `Tensor`) and `mat2` (2D `Tensor`).

Optional values `v1` and `v2` are scalars that multiply `M` and `mat1 * mat2` respectively.

In other words,

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

If `mat1` is a `n × m` matrix, `mat2` a `m × p` matrix, `M` must be a `n × p` matrix.

`torch.addmm(M, mat1, mat2)` returns the result in a new `Tensor`.

`torch.addmm(r, M, mat1, mat2)` puts the result in `r`.

Differences when used as a method

`M:addmm(mat1, mat2)` does `M = M + mat1 * mat2`.

`r:addmm(M, mat1, mat2)` does `r = M + mat1 * mat2`.

`r:addmm(v1, M, v2, mat1, mat2)` does `r = (v1 * M) + (v2 * mat1 * mat2)`.

`M:addmm(v1, v2, mat1, mat2)` does `M = (v1 * M) + (v2 * mat1 * mat2)`.

The last example does not accurately fit into the function signature, and needs a special mention. It changes the function signature to:

`[M] = M:addmm([v1,] [v2,] mat1, mat2)`

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

Batch matrix matrix product of matrices stored in `batch1` and `batch2`, with a reduced add step (all matrix multiplications get accumulated in a single place).

`batch1` and `batch2` must be 3D `Tensor`s each containing the same number of matrices. If `batch1` is a `b × n × m` `Tensor`, `batch2` a `b × m × p` `Tensor`, res will be a `n × p` `Tensor`.

In other words,

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

`torch.addbmm(M, x, y)` puts the result in a new `Tensor`.

`M:addbmm(x, y)` puts the result in `M`, resizing `M` if necessary.

`M:addbmm(beta, M2, alpha, x, y)` puts the result in `M`, resizing `M` if necessary.

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

Batch matrix matrix product of matrices stored in `batch1` and `batch2`, with batch add.

`batch1` and `batch2` must be 3D `Tensor`s each containing the same number of matrices.
If `batch1` is a `b × n × m` `Tensor`, `batch2` a `b × m × p` `Tensor`, res will be a `b × n × p` `Tensor`.

In other words,

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

`torch.baddbmm(M, x, y)` puts the result in a new `Tensor`.

`M:baddbmm(x, y)` puts the result in `M`, resizing `M` if necessary.

`M:baddbmm(beta, M2, alpha, x, y)` puts the result in `M`, resizing `M` if necessary.

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

Matrix vector product of `mat` and `vec`.
Sizes must respect the matrix-multiplication operation: if `mat` is a `n × m` matrix, `vec` must be vector of size `m` and `res` must be a vector of size `n`.

`torch.mv(x, y)` puts the result in a new `Tensor`.

`torch.mv(M, x, y)` puts the result in `M`.

`M:mv(x, y)` puts the result in `M`.

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

Matrix matrix product of `mat1` and `mat2`.
If `mat1` is a `n × m` matrix, `mat2` a `m × p` matrix, `res` must be a `n × p` matrix.

`torch.mm(x, y)` puts the result in a new `Tensor`.

`torch.mm(M, x, y)` puts the result in `M`.

`M:mm(x, y)` puts the result in `M`.

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

Batch matrix matrix product of matrices stored in `batch1` and `batch2`.
`batch1` and `batch2` must be 3D `Tensor`s each containing the same number of matrices.
If `batch1` is a `b × n × m` `Tensor`, `batch2` a `b × m × p` `Tensor`, `res` will be a `b × n × p` `Tensor`.

`torch.bmm(x, y)` puts the result in a new `Tensor`.

`torch.bmm(M, x, y)` puts the result in `M`, resizing `M` if necessary.

`M:bmm(x, y)` puts the result in `M`, resizing `M` if necessary.

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

Outer product of `vec1` and `vec2`.
If `vec1` is a vector of size `n` and `vec2` is a vector of size `m`, then `res` must be a matrix of size `n × m`.

`torch.ger(x, y)` puts the result in a new `Tensor`.

`torch.ger(M, x, y)` puts the result in `M`.

`M:ger(x, y)` puts the result in `M`.

[res] torch.lerp([res,] a, b, weight)

Linear interpolation of two scalars or tensors based on a weight: `res = a + weight * (b - a)`

`torch.lerp(a, b, weight)` puts the result in a new `Tensor` if `a` and `b` are tensors. If `a` and `b` are scalars the functions returns a number.

`torch.lerp(M, a, b, weight)` puts the result in `M`.

`M:lerp(a, b, weight)` puts the result in `M`.

Overloaded operators

It is possible to use basic mathematical operators like `+`, `-`, `/`, `*` and `%` with `Tensor`s.
These operators are provided as a convenience.
While they might be handy, they create and return a new `Tensor` containing the results.
They are thus not as fast as the operations available in the previous section.

Another important point to note is that these operators are only overloaded when the first operand is a `Tensor`.
For example, this will NOT work:

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

Addition and subtraction

You can add a `Tensor` to another one with the `+` operator.
Subtraction is done with `-`.
The number of elements in the `Tensor`s must match, but the sizes do not matter.
The size of the returned `Tensor` will be the size of the first `Tensor`.

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

> = y - x
 1
 1
 1
 1
[torch.DoubleTensor of size 4]
```

A scalar might also be added or subtracted to a `Tensor`.
The scalar needs to be on the right of the operator.

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

Negation

A `Tensor` can be negated with the `-` operator placed in front:

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

Multiplication

Multiplication between two `Tensor` s is supported with the `*` operators.
The result of the multiplication depends on the sizes of the `Tensor` s.

- 1D and 1D: Returns the dot product between the two `Tensor` s (scalar).
- 2D and 1D: Returns the matrix-vector operation between the two `Tensor` s (1D `Tensor`).
- 2D and 2D: Returns the matrix-matrix operation between the two `Tensor` s (2D `Tensor`).

Sizes must be conformant for the corresponding operation.

A `Tensor` might also be multiplied by a scalar.
The scalar might be on the right or left of the operator.

Examples:

```
> 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 -- dot product
40

> = M * x --- matrix-vector
 16
 16
[torch.DoubleTensor of size 2]

> = M * N -- matrix-matrix
 12  12  12  12
 12  12  12  12
[torch.DoubleTensor of size 2x4]
```

Division and Modulo (remainder)

Only the division of a `Tensor` by a scalar is supported with the operator `/`.

Example:

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

Similarly, the remainder of the division of a `Tensor`s elements by a scalar can be obtained with the operator `%`.

Example:

> x = torch.Tensor{{1,2},{3,4}}
> = x % 3
> 1 2
> 0 1
> [torch.Tensor of size 2x2]

Column or row-wise operations (dimension-wise operations)

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

`y = torch.cross(a, b)` returns the cross product of `a` and `b` along the first dimension of length 3.

`y = torch.cross(a, b, n)` returns the cross product of vectors in dimension `n` of `a` and `b`.

`a` and `b` must have the same size, and both `a:size(n)` and `b:size(n)` must be 3.

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

`y = torch.cumprod(x)` returns the cumulative product of the elements of `x`, performing the operation over the last dimension.

`y = torch.cumprod(x, n)` returns the cumulative product of the elements of `x`, performing the operation over dimension `n`.

```
-- 1. cumulative product for a vector
> 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. cumulative product for a matrix
> 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. cumulative product along 2-dim
> 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)` returns the cumulative sum of the elements of `x`, performing the operation over the first dimension.

`y = torch.cumsum(x, n)` returns the cumulative sum of the elements of `x`, performing the operation over dimension `n`.

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

`y = torch.max(x)` returns the single largest element of `x`.

`y, i = torch.max(x, 1)` returns the largest element in each column (across rows) of `x`, and a `Tensor` `i` of their corresponding indices in `x`.

`y, i = torch.max(x, 2)` performs the max operation for each row.

`y, i = torch.max(x, n)` performs the max operation over the dimension `n`.

```
> 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]

> torch.max(x)
1.1993977428735

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

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

> 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)` returns the mean of all elements of `x`.

`y = torch.mean(x, 1)` returns a `Tensor` `y` of the mean of the elements in each column of `x`.

`y = torch.mean(x, 2)` performs the mean operation for each row.

`y = torch.mean(x, n)` performs the mean operation over the dimension `n`.

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

`y = torch.min(x)` returns the single smallest element of `x`.

`y, i = torch.min(x, 1)` returns the smallest element in each column (across rows) of `x`, and a `Tensor` `i` of their corresponding indices in `x`.

`y, i = torch.min(x, 2)` performs the min operation for each row.

`y, i = torch.min(x, n)` performs the min operation over the dimension `n`.

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

Compute the maximum of each pair of values in `tensor1` and `tensor2`.

`c = torch.cmax(a, b)` returns a new `Tensor` containing the element-wise maximum of `a` and `b`.

`a:cmax(b)` stores the element-wise maximum of `a` and `b` in `a`.

`c:cmax(a, b)` stores the element-wise maximum of `a` and `b` in `c`.

```
> 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)

Compute the maximum between each value in `tensor` and `value`.

`c = torch.cmax(a, v)` returns a new `Tensor` containing the maxima of each element in `a` and `v`.

`a:cmax(v)` stores the maxima of each element in `a` and `v` in `a`.

`c:cmax(a, v)` stores the maxima of each element in `a` and `v` in `c`.

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

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

Compute the minimum of each pair of values in `tensor1` and `tensor2`.

`c = torch.cmin(a, b)` returns a new `Tensor` containing the element-wise minimum of `a` and `b`.

`a:cmin(b)` stores the element-wise minimum of `a` and `b` in `a`.

`c:cmin(a, b)` stores the element-wise minimum of `a` and `b` in `c`.

```
> 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)

Compute the minimum between each value in `tensor` and `value`.

`c = torch.cmin(a, v)` returns a new `Tensor` containing the minima of each element in `a` and `v`.

`a:cmin(v)` stores the minima of each element in `a` and `v` in `a`.

`c:cmin(a, v)` stores the minima of each element in `a` and `v` in `c`.

```
> 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)` performs the median operation over the last dimension of `x` (one-before-middle in the case of an even number of elements).

`y, i = torch.median(x, 1)` returns the median element in each column (across rows) of `x`, and a `Tensor` `i` of their corresponding indices in `x`.

`y, i = torch.median(x, 2)` performs the median operation for each row.

`y, i = torch.median(x, n)` performs the median operation over the dimension `n`.

```
> 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)` returns the most frequent element of `x` over its last dimension.

`y, i = torch.mode(x, 1)` returns the mode element in each column (across rows) of `x`, and a `Tensor` `i` of their corresponding indices in `x`.

`y, i = torch.mode(x, 2)` performs the mode operation for each row.

`y, i = torch.mode(x, n)` performs the mode operation over the dimension `n`.

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

`y = torch.kthvalue(x, k)` returns the `k`-th smallest element of `x` over its last dimension.

`y, i = torch.kthvalue(x, k, 1)` returns the `k`-th smallest element in each column (across rows) of `x`, and a `Tensor` `i` of their corresponding indices in `x`.

`y, i = torch.kthvalue(x, k, 2)` performs the `k`-th value operation for each row.

`y, i = torch.kthvalue(x, k, n)` performs the `k`-th value operation over the dimension `n`.

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

`y = torch.prod(x)` returns the product of all elements in `x`.

`y = torch.prod(x, n)` returns a `Tensor` `y` whom size in dimension `n` is 1 and where elements are the product of elements of `x` with respect to dimension `n`.

```
> 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)` returns a `Tensor` `y` where all entries are sorted along the last dimension, in **ascending** order.
It also returns a `Tensor` `i` that provides the corresponding indices from `x`.

`y, i = torch.sort(x, d)` performs the sort operation along a specific dimension `d`.

`y, i = torch.sort(x)` is therefore equivalent to `y, i = torch.sort(x, x:dim())`

`y, i = torch.sort(x, d, true)` performs the sort operation along a specific dimension `d`, in **descending** order.

```
> 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]
```

torch.topk([resval, resind,] x, k, [,dim] [,dir] [,sort])

`y, i = torch.topk(x, k)` returns all `k` smallest elements in `x` over its last dimension including their indices, in unsorted order.

`y, i = torch.topk(x, k, dim)` performs the same operation except over dimension `dim`.

`y, i = torch.topk(x, k, dim, dir)` adds a sorting direction that has the same sense as `torch.sort`; `false` returns the `k` smallest elements in the slice, `true` returns the `k` largest elements in the slice.

`y, i = torch.topk(x, k, dim, dir, true)` specifies that the results in `y` should be sorted with respect to `dir`; by default, the results are potentially unsorted since the computation may be faster, but if sorting is desired, the sort flag may be passed, in which case the results are returned from smallest to `k`-th smallest (`dir == false`) or highest to `k`-th highest (`dir == true`).

The implementation provides no guarantee of the order of selection (indices) among equivalent elements (e.g., topk `k == 2` selection of a vector `{1, 2, 1, 1}`; the values returned could be any pair of `1` entries in the vector).

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

`y = torch.std(x)` returns the standard deviation of the elements of `x`.

`y = torch.std(x, dim)` performs the `std` operation over the dimension `dim`.

`y = torch.std(x, dim, false)` performs the `std` operation normalizing by `n-1` (this is the default).

`y = torch.std(x, dim, true)` performs the `std` operation normalizing by `n` instead of `n-1`.

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

`y = torch.sum(x)` returns the sum of the elements of `x`.

`y = torch.sum(x, 2)` performs the sum operation for each row.

`y = torch.sum(x, n)` performs the sum operation over the dimension `n`.

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

`y = torch.var(x)` returns the variance of the elements of `x`.

`y = torch.var(x, dim)` performs the `var` operation over the dimension `dim`.

`y = torch.var(x, dim, false)` performs the `var` operation normalizing by `n-1` (this is the default).

`y = torch.var(x, dim, true)` performs the `var` operation normalizing by `n` instead of `n-1`.

Matrix-wide operations (`Tensor`-wide operations)

Note that many of the operations in dimension-wise operations can also be used as matrix-wide operations, by just omitting the `dim` parameter.

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

`y = torch.norm(x)` returns the `2`-norm of the `Tensor` `x`.

`y = torch.norm(x, p)` returns the `p`-norm of the `Tensor` `x`.

`y = torch.norm(x, p, dim)` returns the `p`-norms of the `Tensor` `x` computed over the dimension `dim`.

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

Renormalizes the sub-`Tensor`s along dimension `dim` such that they do not exceed norm `maxnorm`.

`y = torch.renorm(x, p, dim, maxnorm)` returns a version of `x` with `p`-norms lower than `maxnorm` over non-`dim` dimensions.
The `dim` argument is not to be confused with the argument of the same name in function `norm`.
In this case, the `p`-norm is measured for each `i`-th sub-`Tensor` `x:select(dim, i)`.
This function is equivalent to (but faster than) the following:

```
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)` returns the equivalent of `x:copy(torch.renorm(x, p, dim, maxnorm))`.

Note: this function is particularly useful as a regularizer for constraining the norm of parameter `Tensor`s.

See Hinton et al. 2012, p. 2.

torch.dist(x, y)

`y = torch.dist(x, y)` returns the `2`-norm of `x - y`.

`y = torch.dist(x, y, p)` returns the `p`-norm of `x - y`.

torch.numel(x)

`y = torch.numel(x)` returns the count of the number of elements in the matrix `x`.

torch.trace(x)

`y = torch.trace(x)` returns the trace (sum of the diagonal elements) of a matrix `x`.
This is equal to the sum of the eigenvalues of `x`.
The returned value `y` is a number, not a `Tensor`.

Convolution Operations

These functions implement convolution or cross-correlation of an input image (or set of input images) with a kernel (or set of kernels).
The convolution function in Torch can handle different types of input/kernel dimensions and produces corresponding outputs.
The general form of operations always remain the same.

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

This function computes 2 dimensional convolutions between `x` and `k`.
These operations are similar to BLAS operations when number of dimensions of input and kernel are reduced by `2`.

- `x` and `k` are 2D: convolution of a single image with a single kernel (2D output). This operation is similar to multiplication of two scalars.
- `x` (`p × m × n`) and `k` (`p × ki × kj`) are 3D: convolution of each input slice with corresponding kernel (3D output).
- `x` (`p × m × n`) 3D, `k` (`q × p × ki × kj`) 4D: convolution of all input slices with the corresponding slice of kernel. Output is 3D (`q × m × n`). This operation is similar to matrix vector product of matrix `k` and vector `x` .

The last argument controls if the convolution is a full (`'F'`) or valid (`'V'`) convolution. The default is **valid** convolution.

```
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'])

This function operates with same options and input/output configurations as `torch.conv2` , but performs cross-correlation of the input with the kernel `k` .

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

This function computes 3 dimensional convolutions between `x` and `k` .
These operations are similar to BLAS operations when number of dimensions of input and kernel are reduced by `3` .

- `x` and `k` are 3D: convolution of a single image with a single kernel (3D output). This operation is similar to multiplication of two scalars.
- `x` (`p × m × n × o`) and `k` (`p × ki × kj × kk`) are 4D: convolution of each input

slice with corresponding kernel (4D output).
- `x` (`p × m × n × o`) 4D, `k` (`q × p × ki × kj × kk`) 5D: convolution of all input slices with the corresponding slice of kernel. Output is 4D `q × m × n × o`. This operation is similar to matrix vector product of matrix `k` and vector `x`.

The last argument controls if the convolution is a full (`'F'`) or valid (`'V'`) convolution. The default is **valid** convolution.

```
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'])

This function operates with same options and input/output configurations as `torch.conv3`, but performs cross-correlation of the input with the kernel `k`.

Eigenvalues, SVD, Linear System Solution

Functions in this section are implemented with an interface to LAPACK libraries.
If LAPACK libraries are not found during compilation step, then these functions will not be available.

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

`X, LU = torch.gesv(B, A)` returns the solution of `AX = B` and `LU` contains `L` and `U` factors for `LU` factorization of `A`.

`A` has to be a square and non-singular matrix (2D `Tensor`).
`A` and `LU` are `m × m`, `X` is `m × k` and `B` is `m × k`.

If `resb` and `resa` are given, then they will be used for temporary storage and returning the result.

- `resa` will contain `L` and `U` factors for `LU` factorization of `A`.
- `resb` will contain the solution `X`.

Note: Irrespective of the original strides, the returned matrices `resb` and `resa` will be transposed, i.e. with strides `1, m` instead of `m, 1`.

```
> 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)` returns the solution of `AX = B` where `A` is upper-triangular.

`A` has to be a square, triangular, non-singular matrix (2D `Tensor`).
`A` and `resa` are `m × m`, `X` and `B` are `m × k`.
(To be very precise: `A` does not have to be triangular and non-singular, rather only its upper or lower triangle will be taken into account and that part has to be non-singular.)

The function has several options:

- `uplo` (`'U'` or `'L'`) specifies whether `A` is upper or lower triangular; the default value is `'U'`.
- `trans` (`'N'` or `'T`') specifies the system of equations: `'N'` for `A * X = B` (no transpose), or `'T'` for `A^T * X = B` (transpose); the default value is `'N'`.
- `diag` (`'N'` or `'U'`) `'U'` specifies that `A` is unit triangular, i.e., it has ones on its diagonal; `'N'` specifies that `A` is not (necessarily) unit triangular; the default value is `'N'`.

If `resb` and `resa` are given, then they will be used for temporary storage and returning the result.
`resb` will contain the solution `X`.

Note: Irrespective of the original strides, the returned matrices `resb` and `resa` will be transposed, i.e. with strides `1, m` instead of `m, 1`.

```
> 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'])

Cholesky Decomposition of 2D `Tensor` `A`.
The matrix `A` has to be a positive-definite and either symmetric or complex Hermitian.

The factorization has the form

```
A = U**T * U,   if UPLO = 'U', or
A = L  * L**T,  if UPLO = 'L',
```

where `U` is an upper triangular matrix and `L` is lower triangular.

The optional character `uplo` = {'U', 'L'} specifies whether the upper or lower triangulardecomposition should be returned. By default, `uplo` = 'U'.

`U = torch.potrf(A, 'U')` returns the upper triangular Cholesky decomposition of `A`.

`L = torch.potrf(A, 'L')` returns the lower triangular Cholesky decomposition of `A`.

If `Tensor` `res` is provided, the resulting decomposition will be stored therein.

```
> 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.pstrf([res, piv,] A [, 'U' or 'L'])

Cholesky factorization with complete pivoting of a real symmetric positive semidefinite 2D `Tensor` `A`.
The matrix `A` has to be a positive semi-definite and symmetric. The factorization has the form

```
P**T * A * P = U**T * U ,  if UPLO = 'U',
P**T * A * P = L  * L**T,  if UPLO = 'L',
```

where `U` is an upper triangular matrix and `L` is lower triangular, and `P` is stored as the vector `piv`. More specifically, `piv` is such that the nonzero entries are `P[piv[k], k] = 1`.

The optional character argument `uplo` = {'U', 'L'} specifies whether the upper or lower triangular decomposition should be returned. By default, `uplo` = 'U'.

`U, piv = torch.sdtrf(A, 'U')` returns the upper triangular Cholesky decomposition of `A`

`L, piv = torch.potrf(A, 'L')` returns the lower triangular Cholesky decomposition of `A`.

If tensors `res` and `piv` (an `IntTensor`) are provided, the resulting decomposition will be stored therein.

```
> 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}})

> U, piv = torch.pstrf(A)
> U
 1.1272  0.4390  0.2112  0.8846  0.1232
 0.0000  0.9750 -0.0354  0.2942 -0.0233
 0.0000  0.0000  0.5915 -0.0961  0.0435
 0.0000  0.0000  0.0000  0.3439 -0.0854
 0.0000  0.0000  0.0000  0.0000  0.0456
[torch.DoubleTensor of size 5x5]

> piv
 1
 3
 5
 2
 4
[torch.IntTensor of size 5]

> Ap = U:t() * U
```

```
> Ap
 1.2705  0.4948  0.2381  0.9971  0.1389
 0.4948  1.1434  0.0582  0.6752  0.0314
 0.2381  0.0582  0.3957  0.1196  0.0526
 0.9971  0.6752  0.1196  0.9966  0.0686
 0.1389  0.0314  0.0526  0.0686  0.0270
[torch.DoubleTensor of size 5x5]

> -- Permute rows and columns
> Ap:indexCopy(1, piv:long(), Ap:clone())
> Ap:indexCopy(2, piv:long(), Ap:clone())
> (Ap - A):norm()
1.5731560566382e-16
```

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

Returns the solution to linear system `AX = B` using the Cholesky decomposition `chol` of 2D `Tensor` `A`.

Square matrix `chol` should be triangular; and, righthand side matrix `B` should be of full rank.

Optional character `uplo` = {'U', 'L'} specifies matrix `chol` as either upper or lower triangular; and, by default, equals 'U'.

If `Tensor` `res` is provided, the resulting decomposition will be stored therein.

```
> 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'])

Returns the inverse of 2D `Tensor` `A` given its Cholesky decomposition `chol`.

Square matrix `chol` should be triangular.

Optional character `uplo` = {'U', 'L'} specifies matrix `chol` as either upper or lower triangular; and, by default, equals 'U'.

If `Tensor` `res` is provided, the resulting inverse will be stored therein.

```
> 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)

Solution of least squares and least norm problems for a full rank `m × n` matrix `A`.

- If `n ≤ m`, then solve `||AX-B||_F`.
- If `n > m`, then solve `min ||X||_F` s.t. `AX = B`.

On return, first `n` rows of `x` matrix contains the solution and the rest contains residual information.
Square root of sum squares of elements of each column of `x` starting at row `n + 1` is the residual for corresponding column.

Note: Irrespective of the original strides, the returned matrices `resb` and `resa` will be transposed, i.e. with strides `1, m` instead of `m, 1`.

```
> 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)` returns eigenvalues and eigenvectors of a symmetric real matrix

`A`.

`A` and `V` are `m × m` matrices and `e` is a `m` dimensional vector.

This function calculates all eigenvalues (and vectors) of `A` such that `A = V diag(e) V'`.

Third argument defines computation of eigenvectors or eigenvalues only.
If it is `'N'`, only eigenvalues are computed.
If it is `'V'`, both eigenvalues and eigenvectors are computed.

Since the input matrix `A` is supposed to be symmetric, only upper triangular portion is used by default.
If the 4th argument is `'L'`, then lower triangular portion is used.

Note: Irrespective of the original strides, the returned matrix `V` will be transposed, i.e. with strides `1, m` instead of `m, 1`.

```
> 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)` returns eigenvalues and eigenvectors of a general real square matrix `A`.

`A` and `V` are `m × m` matrices and `e` is a `m` dimensional vector.

This function calculates all right eigenvalues (and vectors) of `A` such that `A = V diag(e) V'`.

Third argument defines computation of eigenvectors or eigenvalues only.
If it is `'N'`, only eigenvalues are computed.
If it is `'V'`, both eigenvalues and eigenvectors are computed.

The eigen values returned follow LAPACK convention and are returned as complex (real/imaginary) pairs of numbers (`2 * m` dimensional `Tensor`).

Note: Irrespective of the original strides, the returned matrix `V` will be transposed, i.e. with strides `1, m` instead of `m, 1`.

```
> 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)` returns the singular value decomposition of a real matrix `A` of size `n × m` such that `A = USV'*`.

`U` is `n × n`, `S` is `n × m` and `V` is `m × m`.

The last argument, if it is string, represents the number of singular values to be computed. `'S'` stands for *some* and `'A'` stands for *all*.

Note: Irrespective of the original strides, the returned matrix `U` will be transposed, i.e. with strides `1, n` instead of `n, 1`.

```
> 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)

Computes the inverse of square matrix `x`.

`torch.inverse(x)` returns the result as a new matrix.

`torch.inverse(y, x)` puts the result in `y`.

Note: Irrespective of the original strides, the returned matrix `y` will be transposed, i.e. with strides `1, m` instead of `m, 1`.

```
> 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))) -- Max nonzero
2.3092638912203e-14
```

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

Compute a QR decomposition of the matrix `x`: matrices `q` and `r` such that `x = q * r`, with `q` orthogonal and `r` upper triangular.
This returns the thin (reduced) QR factorization.

`torch.qr(x)` returns the Q and R components as new matrices.

`torch.qr(q, r, x)` stores them in existing `Tensor` s `q` and `r`.

Note that precision may be lost if the magnitudes of the elements of `x` are large.

Note also that, while it should always give you a valid decomposition, it may not give you the same one across platforms - it will depend on your LAPACK implementation.

Note: Irrespective of the original strides, the returned matrix `q` will be transposed, i.e. with strides `1, m` instead of `m, 1`.

```
> 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)

This is a low-level function for calling LAPACK directly.
You'll generally want to use `torch.qr()` instead.

Computes a QR decomposition of `a`, but without constructing Q and R as explicit separate matrices.
Rather, this directly calls the underlying LAPACK function `?geqrf` which produces a sequence of 'elementary reflectors'.
See LAPACK documentation for further details.

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

This is a low-level function for calling LAPACK directly.
You'll generally want to use `torch.qr()` instead.

Constructs a Q matrix from a sequence of elementary reflectors, such as that given by `torch.geqrf`.
See LAPACK documentation for further details.

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

Multiply a matrix with `Q` as defined by the elementary reflectors and scalar factors returned by `geqrf`.
This is a low-level function for calling LAPACK directly.
You'll generally want to use `torch.qr()` instead.

- `side` (`'L'` or `'R'`) specifies whether `mat` should be left-multiplied, `mat * Q`, or right-multiplied, `Q * mat`.
- `trans` (`'N'` or `'T'`) specifies whether `Q` should be transposed before being multiplied.

See LAPACK documentation for further details.

Logical Operations on `Tensor`s

These functions implement logical comparison operators that take a `Tensor` as input and another `Tensor` or a number as the comparison target.
They return a `ByteTensor` in which each element is `0` or `1` indicating if the comparison for the corresponding element was `false` or `true` respectively.

torch.lt(a, b)

Implements `<` operator comparing each element in `a` with `b` (if `b` is a number) or each element in `a` with corresponding element in `b`.

torch.le(a, b)

Implements `<=` operator comparing each element in `a` with `b` (if `b` is a number) or each element in `a` with corresponding element in `b`.

torch.gt(a, b)

Implements `>` operator comparing each element in `a` with `b` (if `b` is a number) or each element in `a` with corresponding element in `b`.

torch.ge(a, b)

Implements `>=` operator comparing each element in `a` with `b` (if `b` is a number) or each element in `a` with corresponding element in `b`.

torch.eq(a, b)

Implements `==` operator comparing each element in `a` with `b` (if `b` is a number) or each element in `a` with corresponding element in `b`.

torch.ne(a, b)

Implements `~=` operator comparing each element in `a` with `b` (if `b` is a number) or each element in `a` with corresponding element in `b`.

torch.all(a)

torch.any(a)

Additionally, `any` and `all` logically sum a `ByteTensor` returning `true` if any or all elements are logically true respectively.
Note that logically true here is meant in the C sense (zero is `false`, non-zero is `true`) such as the output of the `Tensor` element-wise logical operations.

```
> 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
```

Tensor

The `Tensor` class is probably the most important class in `Torch`. Almost every package depends on this class. It is ***the*** class for handling numeric data. As with pretty much anything in Torch7, tensors are serializable.

Multi-dimensional matrix

A `Tensor` is a potentially multi-dimensional matrix. The number of dimensions is unlimited that can be created using LongStorage with more dimensions.

Example:

```
 --- creation of a 4D-tensor 4x5x6x2
 z = torch.Tensor(4,5,6,2)
 --- for more dimensions, (here a 6D tensor) one can do:
 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)
```

The number of dimensions of a `Tensor` can be queried by nDimension() or dim(). Size of the `i-th` dimension is returned by size(i). A LongStorage containing all the dimensions can be returned by size().

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

Internal data representation

The actual data of a `Tensor` is contained into a Storage. It can be accessed using `storage()`. While the memory of a `Tensor` has to be contained in this unique `Storage`, it might not be contiguous: the first position used in the `Storage` is given by `storageOffset()` (starting at `1`). And the *jump* needed to go from one element to another element in the `i-th` dimension is given by `stride(i)`. In other words, given a 3D tensor

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

accessing the element `(3,4,5)` can be done by

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

or equivalently (but slowly!)

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

One could say that a `Tensor` is a particular way of *viewing* a `Storage`: a `Storage` only represents a chunk of memory, while the `Tensor` interprets this chunk of memory as having dimensions:

```
x = torch.Tensor(4,5)
s = x:storage()
for i=1,s:size() do -- fill up the Storage
  s[i] = i
end
> x -- s is interpreted by x as a 2D matrix
  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]
```

Note also that in Torch7 ***elements in the same row*** [elements along the **last** dimension] are contiguous in memory for a matrix [tensor]:

```
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  -- element in the last dimension are contiguous!
[torch.LongStorage of size 2]
```

This is exactly like in C (and not `Fortran`).

Tensors of different types

Actually, several types of `Tensor` exists:

```
ByteTensor -- contains unsigned chars
CharTensor -- contains signed chars
ShortTensor -- contains shorts
IntTensor -- contains ints
LongTensor -- contains longs
FloatTensor -- contains floats
DoubleTensor -- contains doubles
```

Most numeric operations are implemented *only* for `FloatTensor` and `DoubleTensor`. Other Tensor types are useful if you want to save memory space.

Default Tensor type

For convenience, *an alias* `torch.Tensor` is provided, which allows the user to write type-independent scripts, which can then ran after choosing the desired Tensor type with a call like

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

See torch.setdefaulttensortype for more details.
By default, the alias "points" on `torch.DoubleTensor`.

Efficient memory management

All tensor operations in this class do *not* make any memory copy. All these methods transform the existing tensor, or return a new tensor referencing *the same storage*. This magical behavior is internally obtained by good usage of the stride() and storageOffset(). Example:

```
x = torch.Tensor(5):zero()
> x
0
0
0
0
0
[torch.DoubleTensor of dimension 5]
> x:narrow(1, 2, 3):fill(1) -- narrow() returns a Tensor
                            -- referencing the same Storage as x
> x
 0
 1
 1
 1
 0
[torch.Tensor of dimension 5]
```

If you really need to copy a `Tensor`, you can use the copy() method:

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

Or the convenience method

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

We now describe all the methods for `Tensor`. If you want to specify the Tensor type, just replace `Tensor` by the name of the Tensor variant (like `CharTensor`).

Tensor constructors

Tensor constructors, create new Tensor object, optionally, allocating new memory. By default the elements of a newly allocated memory are not initialized, therefore, might contain arbitrary numbers. Here are several ways to construct a new `Tensor`.

torch.Tensor()

Returns an empty tensor.

torch.Tensor(tensor)

Returns a new tensor which reference the same Storage than the given `tensor`. The size, stride, and storage offset are the same than the given tensor.

The new `Tensor` is now going to "view" the same storage as the given `tensor`. As a result, any modification in the elements of the `Tensor` will have a impact on the elements of the given `tensor`, and vice-versa. No memory copy!

```
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 -- elements of x are the same as y!
0 0 0 0 0
0 0 0 0 0
[torch.DoubleTensor of dimension 2x5]
```

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

Create a tensor up to 4 dimensions. The tensor size will be `sz1 x sz2 x sx3 x sz4`.

torch.Tensor(sizes, [strides])

Create a tensor of any number of dimensions. The LongStorage `sizes` gives the size in each dimension of the tensor. The optional LongStorage `strides` gives the jump necessary to go from one element to the next one in the each dimension. Of course, `sizes` and `strides` must have the same number of elements. If not given, or if some elements of `strides` are *negative*, the stride() will be computed such that the tensor is as contiguous as possible in memory.

Example, create a 4D 4x4x3x2 tensor:

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

Playing with the strides can give some interesting things:

```
x = torch.Tensor(torch.LongStorage({4}),
torch.LongStorage({0})):zero() -- zeroes the tensor
x[1] = 1 -- all elements point to the same address!
> x
 1
```

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

Note that *negative strides are not allowed*, and, if given as argument when constructing the Tensor, will be interpreted as //choose the right stride such that the Tensor is contiguous in memory//.

Note *this method cannot be used to create* `torch.LongTensor`s. The constructor from a storage will be used:

```
a = torch.LongStorage({1,2}) -- We have a torch.LongStorage
containing the values 1 and 2
-- General case for TYPE ~= Long, e.g. for TYPE = Float:
b = torch.FloatTensor(a)
-- Creates a new torch.FloatTensor with 2 dimensions, the first of
size 1 and the second of size 2
> b:size()
 1
 2
[torch.LongStorage of size 2]

-- Special case of torch.LongTensor
c = torch.LongTensor(a)
-- Creates a new torch.LongTensor that uses a as storage and thus
contains the values 1 and 2
> c
 1
 2
[torch.LongTensor of size 2]
```

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

Returns a tensor which uses the existing Storage `storage`, starting at position `storageOffset` (>=1). The size of each dimension of the tensor is given by the LongStorage `sizes`.

If only `storage` is provided, it will create a 1D Tensor viewing the all Storage.

The jump necessary to go from one element to the next one in each dimension is given by the optional argument LongStorage `strides`. If not given, or if some elements of `strides` are negative, the stride() will be computed such that the tensor is as contiguous as possible in memory.

Any modification in the elements of the `Storage` will have an impact on the elements of the new `Tensor`, and vice-versa. There is no memory copy!

```
-- creates a storage with 10 elements
s = torch.Storage(10):fill(1)

-- we want to see it as a 2x5 tensor
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 -- the storage contents have been modified
 0
 0
 0
 0
 0
 0
 0
 0
 0
 0
[torch.DoubleStorage of size 10]
```

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

Convenience constructor (for the previous constructor) assuming a number of dimensions inferior or equal to 4. `szi` is the size in the `i-th` dimension, and `sti` is the stride in the `i-th` dimension.

torch.Tensor(table)

The argument is assumed to be a Lua array of numbers. The constructor returns a new Tensor of the size of the table, containing all the table elements. The table might be multi-dimensional.

Example:

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

A note on function calls

The rest of this guide will present many functions that can be used to manipulate tensors. Most functions have been
defined so that they can be called flexibly, either in an object-oriented "method call" style i.e. `src:function(...)`
or a more "functional" style `torch.function(src, ...)`, where `src` is a tensor. Note that these different invocations
may differ in whether they modify the tensor in-place, or create a new tensor. Additionally, some functions can be
called in the form `dst:function(src, ...)` which usually suggests that the result of the operation on the `src` tensor
will be stored in the tensor `dst`. Further details are given in the individual function definitions, below, but it
should be noted that the documentation is currently incomplete in this regard, and readers are encouraged to experiment
in an interactive session.

Cloning

[Tensor] clone()

Returns a clone of a tensor. The memory is copied.

```
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]

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

-- fill up y with 1
y:fill(1)
> y
 1
 1
 1
 1
 1
[torch.DoubleTensor of dimension 5]

-- the contents of x were not changed:
> x
 1
 2
 3
 4
 5
[torch.DoubleTensor of dimension 5]
```

[Tensor] contiguous

- If the given Tensor contents are contiguous in memory, returns the exact same Tensor (no memory copy).
- Otherwise (*not contiguous in memory*), returns a clone (memory *copy*).

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

-- x is contiguous, so y points to the same thing
y = x:contiguous():fill(2)
> y
 2  2  2
 2  2  2
[torch.DoubleTensor of dimension 2x3]

-- contents of x have been changed
> x
 2  2  2
 2  2  2
[torch.DoubleTensor of dimension 2x3]

-- x:t() is not contiguous, so z is a clone
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]

-- contents of x have not been changed
> x
 2  2  2
 2  2  2
[torch.DoubleTensor of dimension 2x3]
```

[Tensor or string] type(type)

If `type` is `nil`, returns a string containing the type name of the given tensor.

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

If `type` is a string describing a Tensor type, and is equal to the given tensor typename, returns the exact same tensor (//no memory copy//).

```
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]

-- zero y contents
y:zero()

-- contents of x have been changed
> x
 0
 0
 0
[torch.DoubleTensor of dimension 3]
```

If `type` is a string describing a Tensor type, different from the type name of the given Tensor, returns a new Tensor of the specified type, whose contents corresponds to the contents of the original Tensor, casted to the given type (//memory copy occurs, with possible loss of precision//).

```
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)

Convenience method for the type method. Equivalent to

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

[boolean] isTensor(object)

Returns `true` iff the provided `object` is one of the `torch.*Tensor` types.

```
> 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()

Convenience methods for the type method. For e.g.,

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

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


-- is equivalent to calling int()
> x:int()
 3
 3
 3
[torch.IntTensor of dimension 3]
```

Querying the size and structure

[number] nDimension()

Returns the number of dimensions in a `Tensor`.

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

```
2
```

[number] dim()

Same as nDimension().

[number] size(dim)

Returns the size of the specified dimension `dim`. Example:

```
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) -- gets the number of columns
5
```

[LongStorage] size()

Returns a LongStorage containing the size of each dimension of the tensor.

```
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

Same as size() method.

[number] stride(dim)

Returns the jump necessary to go from one element to the next one in the specified dimension `dim`. Example:

```
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]

-- elements in a row are contiguous in memory
> x:stride(2)
1

-- to go from one element to the next one in a column
-- we need here to jump the size of the row
> x:stride(1)
5
```

Note also that in `Torch` *elements in the same row* [elements along the **last** dimension] are contiguous in memory for a matrix [tensor].

[LongStorage] stride()

Returns the jump necessary to go from one element to the next one in each dimension. Example:

```
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 -- elements are contiguous in a row [last dimension]
[torch.LongStorage of size 2]
```

Note also that in `Torch` *elements in the same row* [elements along the **last** dimension] are contiguous in memory for a matrix [tensor].

[Storage] storage()

Returns the Storage used to store all the elements of the `Tensor`.
Basically, a `Tensor` is a particular way of *viewing* a `Storage`.

```
x = torch.Tensor(4,5)
s = x:storage()
for i=1,s:size() do -- fill up the Storage
  s[i] = i
end

> x -- s is interpreted by x as a 2D matrix
  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()

Returns `true` iff the elements of the `Tensor` are contiguous in memory.

```
-- normal tensors are contiguous in memory
x = torch.randn(4,5)
> x:isContiguous()
true

-- y now "views" the 3rd column of x
-- the storage of y is the same than x
-- so the memory cannot be contiguous
y = x:select(2, 3)
> y:isContiguous()
false

-- indeed, to jump to one element to
-- the next one, the stride is 5
> y:stride()
 5
[torch.LongStorage of size 1]
```

[boolean] isSize(storage)

Returns `true` iff the dimensions of the `Tensor` match the elements of the `storage`.

```
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)

Returns `true` iff the dimensions of the `Tensor` and the argument `Tensor` are exactly the same.

```
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()

Returns the number of elements of a tensor.

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

[number] storageOffset()

Return the first index (starting at 1) used in the tensor's storage.

Querying elements

Elements of a tensor can be retrieved with the `[index]` operator.

If `index` is a number, `[index]` operator is equivalent to a `select(1, index)`. If the tensor has more than one dimension, this operation returns a slice of the tensor that shares the same underlying storage. If the tensor is a 1D tensor, it returns the value at `index` in this tensor.

If `index` is a table, the table must contain *n* numbers, where *n* is the number of dimensions of the Tensor. It will return the element at the given position.

In the same spirit, `index` might be a LongStorage, specifying the position (in the Tensor) of the element to be retrieved.

If `index` is a `ByteTensor` in which each element is 0 or 1 then it acts as a selection mask used to extract a subset of the original tensor. This is particularly useful with logical operators like `torch.le`.

Example:

```
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] -- returns row 2
 4
 5
 6
[torch.DoubleTensor of dimension 3]

> x[2][3] -- returns row 2, column 3
6

> x[{2,3}] -- another way to return row 2, column 3
6

> x[torch.LongStorage{2,3}] -- yet another way to return row 2,
column 3
6

> x[torch.le(x,3)] -- torch.le returns a ByteTensor that acts as a
mask
 1
 2
 3
[torch.DoubleTensor of dimension 3]
```

Referencing a tensor to an existing tensor or chunk of memory

A `Tensor` being a way of *viewing* a Storage, it is possible to "set" a `Tensor` such that it views an existing Storage.

Note that if you want to perform a set on an empty `Tensor` like

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

you might want in that case to use one of the equivalent constructor.

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

[self] set(tensor)

The `Tensor` is now going to "view" the same storage as the given `tensor`. As the result, any modification in the elements of the `Tensor` will have an impact on the elements of the given `tensor`, and vice-versa. This is an efficient method, as there is no memory copy!

```
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 -- elements of x are the same than y!
 0 0 0 0 0
 0 0 0 0 0
[torch.DoubleTensor of dimension 2x5]
```

[boolean] isSetTo(tensor)

Returns true iff the `Tensor` is set to the argument `Tensor`. Note: this is only true if the tensors are the same size, have the same strides and share the same storage and offset.

```
x = torch.Tensor(2,5)
y = torch.Tensor()
> y:isSetTo(x)
 false
> y:set(x)
> y:isSetTo(x)
  true
> y:t():isSetTo(x)
  false -- x and y have different strides
```

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

The `Tensor` is now going to "view" the given `storage`, starting at position `storageOffset` (>=1) with the given dimension `sizes` and the optional given `strides`. As the result, any modification in the elements of the `Storage` will have a impact on the elements of the `Tensor`, and vice-versa. This is an efficient method, as there is no memory copy!

If only `storage` is provided, the whole storage will be viewed as a 1D Tensor.

```
-- creates a storage with 10 elements
s = torch.Storage(10):fill(1)

-- we want to see it as a 2x5 tensor
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 -- the storage contents have been modified
 0
 0
 0
 0
 0
 0
 0
 0
 0
 0
[torch.DoubleStorage of size 10]
```

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

This is a "shortcut" for previous method.
It works up to 4 dimensions. `szi` is the size of the `i` -th dimension of the tensor.
`sti` is the stride in the `i` -th dimension.

Copying and initializing

[self] copy(tensor)

Replace the elements of the `Tensor` by copying the elements of the given `tensor` . The number of elements must match, but the
sizes might be different.

```
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]
```

If a different type of `tensor` is given, then a type conversion occurs, which, of course, might result in loss of precision.

[self] fill(value)

Fill the tensor with the given `value`.

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

[self] zero()

Fill the tensor with zeros.

```
> torch.Tensor(4):zero()
 0
 0
 0
 0
```

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

Resizing

When resizing to a larger size, the underlying Storage is resized to fit all the elements of the `Tensor`.

When resizing to a smaller size, the underlying Storage is not resized.

Important note: the content of a `Tensor` after resizing is *undetermined* as strides might have been completely changed. In particular, *the elements of the resized tensor are contiguous in memory*.

[self] resizeAs(tensor)

Resize the `tensor` as the given `tensor` (of the same type).

[self] resize(sizes)

Resize the `tensor` according to the given LongStorage `sizes`.

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

Convenience method of the previous method, working for a number of dimensions up to 4.

Extracting sub-tensors

Each of these methods returns a `Tensor` which is a sub-tensor of the given tensor.

For methods `narrow`, `select` and `sub` the returned tensor *shares the same* `Storage` as the original. Hence, any modification in the memory of the sub-tensor will have an impact on the primary tensor, and vice-versa. These methods are very fast, as they do not involve any memory copy.

For all other methods in this section such as `index`, `indexCopy` etc., since you cannot extract a shared subtensor (technically), a new tensor is returned. If you make changes in this new tensor, they are not reflected in the original tensor.

[self] narrow(dim, index, size)

Returns a new `Tensor` which is a narrowed version of the current one: the dimension `dim` is narrowed
from `index` to `index+size-1`.

```
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) -- narrow dimension 1 from index 2 to index
2+3-1
y:fill(1) -- fill with 1
> 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 -- memory in x has been modified!
 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]])

This method is equivalent to do a series of narrow up to the first 4 dimensions. It returns a new `Tensor` which is a sub-tensor going from index `dimis` to `dimie` in the `i` -th dimension. Negative values are interpreted index starting from the end: `-1` is the last index, `-2` is the index before the last index, ...

```
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) -- y is sub-tensor of x:
> y                    -- dimension 1 starts at index 2, ends at
index 4
 1  1  1  1  1  1
 1  1  1  1  1  1
 1  1  1  1  1  1
[torch.DoubleTensor of dimension 3x6]

> x                    -- x has been modified!
 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) -- we now take a new sub-tensor
> z                        -- dimension 1 starts at index 2, ends
at index 4
                           -- dimension 2 starts at index 3, ends
at index 4
 2  2
 2  2
 2  2
[torch.DoubleTensor of dimension 3x2]
```

```
> x                            -- x has been modified
 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                            -- y has been modified
 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)          -- negative values = bounds
 2  2
[torch.DoubleTensor of dimension 1x2]
```

[Tensor] select(dim, index)

Returns a new `Tensor` which is a tensor slice at the given `index` in the dimension `dim`. The returned tensor has one less dimension: the dimension `dim` is removed. As a result, it is not possible to `select()` on a 1D tensor.

Note that "selecting" on the first dimension is equivalent to use the [] operator

```
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) -- select row 2 and fill up
> 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) -- select column 5 and fill up
> 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} }]

The indexing operator [] can be used to combine narrow/sub and select in a concise and efficient way. It can also be used to copy, and fill (sub) tensors.

This operator also works with an input mask made of a `ByteTensor` with 0 and 1 elements, e.g with a logical operator.

```
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 -- sets element at (i=1,j=3) to 1
> 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  -- sets a slice of 3 elements to 2
> 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 -- sets the full 4th column to -1
> 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) -- copy a 1D tensor to a slice of x
> 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 -- sets all negative elements to -2 via a
mask
> 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)

Returns a new `Tensor` which indexes the original `Tensor` along dimension `dim`
using the entries in `torch.LongTensor` `index`.
The returned `Tensor` has the same number of dimensions as the original `Tensor`.
The returned `Tensor` does **not** use the same storage as the original `Tensor` – see below for storing the result
in an existing `Tensor`.

```
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})
> 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
 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]
```

Note the explicit `index` function is different than the indexing operator `[]`.
The indexing operator `[]` is a syntactic shortcut for a series of select and narrow operations, therefore it always returns a new view on the original tensor that shares the same storage. However, the explicit `index` function can not use the same storage.

It is possible to store the result into an existing Tensor with `result:index(source, ...)`:

```
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)

Copies the elements of `tensor` into the original tensor by selecting the indices in the order given in `index`. The shape of `tensor` must exactly match the elements indexed or an error will be thrown.

```
> 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)
> 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] indexAdd(dim, index, tensor)

Accumulate the elements of `tensor` into the original tensor by adding to the indices in the order
given in `index`. The shape of `tensor` must exactly match the elements indexed or an error will be thrown.

```
Example 1

> x
-2.1742  0.5688 -1.0201  0.1383  1.0504
 0.0970  0.2169  0.1324  0.9553 -1.9518
-0.7607  0.8947  0.1658 -0.2181 -2.1237
-1.4099  0.2342  0.4549  0.6316 -0.2608
 0.0349  0.4713  0.0050  0.1677  0.2103
[torch.DoubleTensor of size 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:indexAdd(2,torch.LongTensor{5,1},z)
> x
-4.1742  0.5688 -1.0201  0.1383  0.0504
-1.9030  0.2169  0.1324  0.9553 -2.9518
-2.7607  0.8947  0.1658 -0.2181 -3.1237
-3.4099  0.2342  0.4549  0.6316 -1.2608
-1.9651  0.4713  0.0050  0.1677 -0.7897
[torch.DoubleTensor of size 5x5]

Example 2

> a = torch.range(1, 5)
> a
 1
 2
 3
 4
 5
[torch.DoubleTensor of size 5]

> a:indexAdd(1, torch.LongTensor{1, 1, 3, 3}, torch.range(1, 4))
> a
  4
  2
 10
  4
  5
[torch.DoubleTensor of size 5]
```

[Tensor] indexFill(dim, index, val)

Fills the elements of the original `Tensor` with value `val` by selecting the indices in the order given in `index`.

```
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)
> 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)

Creates a new `Tensor` from the original tensor by gathering a number of values from each “row”, where the rows are along the dimension `dim`. The values in a `LongTensor`, passed as `index`,
specify which values to take from each row. Specifically, the resulting `Tensor`, which will have the same size as
the `index` tensor, is given by

```
-- 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.
```

where `src` is the original `Tensor`.

The same number of values are selected from each row, and the same value cannot be selected from a row more than once. The values in the `index` tensor must not be larger than the length of the row, that is they must be between 1 and `src:size(dim)` inclusive. It can be somewhat confusing to ensure that the `index` tensor has the correct shape.
Viewed pictorially:

Numerically, to give an example, if `src` has size `n x m x p x q`, we are gathering along `dim = 3`, and we wish to gather `k` elements from each row (where `k <= p`) then `index` must have size `n x m x k x q`.

It is possible to store the result into an existing Tensor with `result:gather(src, ...)`.

```
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)

Writes all values from tensor `src` or the scalar `val` into `self` at the specified indices. The indices are specified
with respect to the given dimension, `dim`, in the manner described in gather. Note that, as for gather, the values of index must be between 1 and `self:size(dim)` inclusive and all values in a row along the
specified dimension must be unique.

```
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)

Returns a new Tensor which contains all elements aligned to a `1` in the corresponding `mask`. This `mask` is a `torch.ByteTensor` of zeros and ones. The `mask` and `Tensor` must have the same number of elements. The resulting Tensor will
be a 1D tensor of the same type as `Tensor` having size `mask:sum()`.

```
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
```

Note how the dimensions of the above `x`, `mask` and `y` do not match.
Also note how an existing tensor `z` can be used to store the results.

[Tensor] maskedCopy(mask, tensor)

Copies the elements of `tensor` into `mask` locations of itself. The masked elements are those elements having a
corresponding `1` in the `mask` Tensor. This `mask` is a `torch.ByteTensor`
of zeros and ones. The destination `Tensor` and the `mask` Tensor should have the same number of elements.
The source `tensor` should have at least as many elements as the number of 1s in the `mask`.

```
x = torch.Tensor({0, 0, 0, 0})
```

```
mask = torch.ByteTensor({0, 1, 0, 1})
y = torch.Tensor({10, 20})
x:maskedCopy(mask,y)
print(x)

  0
 10
  0
 20
[torch.DoubleTensor of size 4]
```

```
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]
```

Note how the dimensions of the above `x`, `mask` and `y' do not match, but the number of elements do.

[Tensor] maskedFill(mask, val)

Fills the masked elements of itself with value `val`. The masked elements are those elements having a corresponding `1` in the `mask` Tensor. This `mask` is a `torch.ByteTensor` of zeros and ones. The `mask` and `Tensor` must have the same number of elements.

```
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]
```

Note how the dimensions of the above `x` and `mask` do not match, but the number of elements do.

Search

Each of these methods returns a `LongTensor` corresponding to the indices of the given search operation.

[LongTensor] nonzero(tensor)

Finds and returns a `LongTensor` corresponding to the *subscript* indices of all non-zero elements in `tensor`.

Note that torch uses the first argument on dispatch to determine the return type. Since the first argument is any `torch.TensorType`, but the return type is always `torch.LongTensor`, the function call `torch.nonzero(torch.LongTensor(), tensor)` does not work. However, `tensor.nonzero(torch.LongTensor(), tensor)` does work.

```
> 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]
```

Expanding/Replicating/Squeezing Tensors

These methods returns a Tensor which is created by replications of the original tensor.

[result] expand([result,] sizes)

`sizes` can either be a `torch.LongStorage` or numbers. Expanding a tensor does not allocate new memory, but only creates a new view on the existing tensor where singleton dimensions can be expanded to multiple ones by setting the `stride` to 0. Any dimension that has size 1 can be expanded to arbitrary value without any new memory allocation. Attempting to expand along a dimension that does not have size 1 will result in an error.

```
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)

This is equivalent to `self:expand(tensor:size())`

[Tensor] repeatTensor([result,] sizes)

`sizes` can either be a `torch.LongStorage` or numbers. Repeating a tensor allocates new memory, unless `result` is provided, in which case its memory is resized. `sizes` specify the number of times the tensor is repeated in each dimension.

“`lua
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])

Removes all singleton dimensions of the tensor.
If `dim` is given, squeezes only that particular dimension of the tensor.

“`lua
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]

“`

Manipulating the tensor view

Each of these methods returns a `Tensor` which is another way of viewing the `Storage` of the given tensor. Hence, any modification in the memory of the sub-tensor will have an impact on the primary tensor, and vice-versa.

These methods are very fast, because they do not involve any memory copy.

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

Creates a view with different dimensions of the storage associated with `tensor`. If `result` is not passed, then a new tensor is returned, otherwise its storage is made to point to storage of `tensor`.

`sizes` can either be a `torch.LongStorage` or numbers. If one of the dimensions is -1, the size of that dimension is inferred from the rest of the elements.

```
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)

Creates a view with the same dimensions as `template` of the storage associated with `tensor`. If `result` is not passed, then a new tensor is returned, otherwise its storage is made to point to storage of `tensor`.

```
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)

Returns a tensor where dimensions `dim1` and `dim2` have been swapped. For 2D tensors, the convenience method of t() is available.

```
x = torch.Tensor(3,4):zero()
x:select(2,3):fill(7) -- fill column 3 with 7
> x
 0  0  7  0
 0  0  7  0
 0  0  7  0
[torch.DoubleTensor of dimension 3x4]

y = x:transpose(1,2) -- swap dimension 1 and 2
> y
 0  0  0
 0  0  0
 7  7  7
 0  0  0
[torch.DoubleTensor of dimension 4x3]

y:select(2, 3):fill(8) -- fill column 3 with 8
> y
 0  0  8
 0  0  8
 7  7  8
 0  0  8
[torch.DoubleTensor of dimension 4x3]

> x -- contents of x have changed as well
 0  0  7  0
 0  0  7  0
 8  8  8  8
[torch.DoubleTensor of dimension 3x4]
```

[Tensor] t()

Convenience method of transpose() for 2D
tensors. The given tensor must be 2 dimensional. Swap dimensions 1 and 2.

```
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)

Generalizes the function transpose() and can be used
as a convenience method replacing a sequence of transpose() calls.
Returns a tensor where the dimensions were permuted according to the permutation
given by (dim1, dim2, ... , dimn). The permutation must be specified fully, i.e.
there must be as many parameters as the tensor has dimensions.

```
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) -- equivalent to y =
x:transpose(1,3):transpose(1,2)
```

```
> y:size()
 4
 2
 3
 5
[torch.LongStorage of size 4]
```

[Tensor] unfold(dim, size, step)

Returns a tensor which contains all slices of size `size` in the dimension `dim`. Step between two slices is given by `step`.

If `sizedim` is the original size of dimension `dim`, the size of dimension `dim` in the returned tensor will be `(sizedim - size) / step + 1`

An additional dimension of size `size` is appended in the returned tensor.

```
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]
```

Applying a function to a tensor

These functions apply a function to each element of the tensor on which called the method (self). These methods are much faster than using a `for` loop in `Lua`. The results is stored in `self` (if the function returns something).

[self] apply(function)

Apply the given function to all elements of self.

The function takes a number (the current element of the tensor) and might return a number, in which case it will be stored in self.

Examples:

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

z:apply(math.sin) -- apply the sin function
> 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) -- compute the sum of the elements
> sum
1.9552094821074

> z:sum() -- it is indeed correct!
1.9552094821074
```

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

Apply the given function to all elements of self and `tensor`. The number of elements of both tensors
must match, but sizes do not matter.

The function takes two numbers (the current element of self and `tensor`) and might return a number, in which case it will be stored in self.

Example:

```
x = torch.Tensor(3,3)
y = torch.Tensor(9)
i = 0
x:apply(function() i = i + 1; return i end) -- fill-up x
i = 0
y:apply(function() i = i + 1; return i end) -- fill-up y
> 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) -- element-wise
multiplication
> x
  1   4   9
 16  25  36
 49  64  81
[torch.DoubleTensor of dimension 3x3]
```

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

Apply the given function to all elements of self, `tensor1` and `tensor2`. The number of elements of all tensors
must match, but sizes do not matter.

The function takes three numbers (the current element of self, `tensor1` and `tensor2`) and might return
a number, in which case it will be stored in self.

Example:

```
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]
```

Dividing a tensor into a table of tensors

These functions divide a Tensor into a table of Tensors.

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

Splits Tensor `tensor` along dimension `dim` into a `result` table of Tensors of size `size` (a number) or less (in the case of the last Tensor). The sizes of the non- `dim` dimensions remain unchanged. Internally, a series of narrows are performed along dimensions `dim`. Argument `dim` defaults to 1.

If `result` is not passed, then a new table is returned, otherwise it is emptied and reused.

Example:

```
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])

Splits Tensor `tensor` into `n` chunks of approximately equal size along dimensions `dim` and returns these as a `result` table of Tensors. Argument `dim` defaults to 1.

This function uses split internally:

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

Example:

```
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 access

These functions expose Torch's Tensor and Storage data structures, through LuaJIT FFI.
This allows extremely fast access to Tensors and Storages, all from Lua.

[result] data(tensor, [asnumber])

Returns a LuaJIT FFI pointer to the raw data of the tensor.
If `asnumber` is true, then returns the pointer as a `intptr_t` cdata that you can transform to a plain lua number with `tonumber()`.

Accessing the raw data of a Tensor like this is extremely efficient, in fact, it's almost as fast as C in lots of cases.

Example:

```
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]
```

WARNING: bear in mind that accessing the raw data like this is dangerous, and should only be done on contiguous tensors (if a tensor is not contiguous, then you have to use its size and stride information). Making sure a tensor is contiguous is easy:

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

-- it would be unsafe to work with torch.data(t_noncontiguous)
t_transposed_and_contiguous = t_noncontiguous:contiguous()

-- it is now safe to work with the raw pointer
data = torch.data(t_transposed_and_contiguous)
```

Last, the pointer can be returned as a plain `intptr_t` cdata. This can be useful to share pointers between threads (warning: this is dangerous, as the second tensor doesn't increment the reference counter on the storage. If the first tensor gets freed, then the data of the second tensor becomes a dangling pointer):

```
t = torch.randn(10)
p = tonumber(torch.data(t,true))
s = torch.Storage(10, p)
tt = torch.Tensor(s)
-- tt and t are a view on the same data.
```

[result] cdata(tensor, [asnumber])

Returns a LuaJIT FFI pointer to the C structure of the tensor.
Use this with caution, and look at FFI.lua
for the members of the tensor

Reference counting

Tensors are reference-counted. It means that each time an object (C or the Lua state) need to keep a reference over a tensor, the corresponding tensor reference counter will be increased. The reference counter is decreased when the object does not need the tensor anymore.

These methods should be used with extreme care. In general, they should never be called, except if you know what you are doing, as the handling of references is done automatically. They can be useful in threaded environments. Note that these methods are atomic operations.

retain()

Increment the reference counter of the tensor.

free()

Decrement the reference counter of the tensor. Free the tensor if the counter is at 0.

Torch utility functions

These functions are used in all Torch package for creating and handling classes.
The most interesting function is probably `torch.class()` which allows
the user to create easily new classes. `torch.typename()` might
also be interesting to check what is the class of a given *Torch7* object.

The other functions are for more advanced users.

[metatable] torch.class(name, [parentName], [module])

Creates a new `Torch` class called `name`. If `parentName` is provided, the class will inherit `parentName` methods. A class is a table which has a particular metatable.

If `module` is not provided and if `name` is of the form `package.className` then the class `className` will be added to the specified `package`. In that case, `package` has to be a valid (and already loaded) package. If `name` does not contain any `.`, then the class will be defined in the global environment.

If `module` is provided table, the class will be defined in this table at key `className`.

One [or two] (meta)tables are returned. These tables contain all the method provided by the class [and its parent class if it has been provided]. After a call to `torch.class()` you have to fill-up properly the metatable.

After the class definition is complete, constructing a new class `name` will be achieved by a call to `name()`.
This call will first call the method `lua__init()` if it exists, passing all arguments of `name()`.

```
-- for naming convenience
do
   --- creates a class "Foo"
   local Foo = torch.class('Foo')

   --- the initializer
   function Foo:__init()
```

```lua
      self.contents = 'this is some text'
   end

   --- a method
   function Foo:print()
      print(self.contents)
   end

   --- another one
   function Foo:bip()
      print('bip')
   end

end

--- now create an instance of Foo
foo = Foo()

--- try it out
foo:print()

--- create a class torch.Bar which
--- inherits from Foo
do
   local Bar, parent = torch.class('torch.Bar', 'Foo')

   --- the initializer
   function Bar:__init(stuff)
      --- call the parent initializer on ourself
      parent.__init(self)

      --- do some stuff
      self.stuff = stuff
   end

   --- a new method
   function Bar:boing()
      print('boing!')
   end

   --- override parent's method
   function Bar:print()
      print(self.contents)
      print(self.stuff)
   end
```

```
end

--- create a new instance and use it
bar = torch.Bar('ha ha!')
bar:print() -- overrided method
bar:boing() -- child method
bar:bip()   -- parent's method
```

For advanced users, it is worth mentionning that `torch.class()` actually calls `torch.newmetatable()` with a particular constructor. The constructor creates a Lua table and set the right metatable on it, and then calls `lua__init()` if it exists in the metatable. It also sets a factory field `lua__factory` such that it is possible to create an empty object of this class.

[string] torch.type(object)

Checks if `object` has a metatable. If it does, and if it corresponds to a `Torch` class, then returns a string containing the name of the class. Otherwise, it returns the Lua `type(object)` of the object. Unlike `torch.typename()`, all outputs are strings:

```
> torch.type(torch.Tensor())
torch.DoubleTensor
> torch.type({})
table
> torch.type(7)
number
```

[string] torch.typename(object)

Checks if `object` has a metatable. If it does, and if it corresponds to a `Torch` class, then returns a string containing the name of the class. Returns `nil` in any other cases.

```
> torch.typename(torch.Tensor())
torch.DoubleTensor
> torch.typename({})
```

```
> torch.typename(7)
```

A Torch class is a class created with `torch.class()` or `torch.newmetatable()`.

[userdata] torch.typename2id(string)

Given a Torch class name specified by `string`, returns a unique corresponding id (defined by a `lightuserdata` pointing on the internal structure of the class). This might be useful to do a *fast* check of the class of an object (if used with `torch.id()`), avoiding string comparisons.

Returns `nil` if `string` does not specify a Torch object.

[userdata] torch.id(object)

Returns a unique id corresponding to the `class` of the given *Torch7* object. The id is defined by a `lightuserdata` pointing on the internal structure of the class.

Returns `nil` if `object` is not a Torch object.

This is different from the `object` id returned by `torch.pointer()`.

[boolean] isTypeOf(object, typeSpec)

Checks if a given `object` is an instance of the type specified by `typeSpec`. `typeSpec` can be a string (including a `string.find` pattern) or the constructor object for a Torch class. This function traverses up the class hierarchy, so if b is an instance of B which is a subclass of A, then `torch.isTypeOf(b, B)` and `torch.isTypeOf(b, A)` will both return `true`.

[table] torch.newmetatable(name, parentName, constructor)

Register a new metatable as a Torch type with the given string `name`. The new metatable is returned.

If the string `parentName` is not `nil` and is a valid Torch type (previously created by `torch.newmetatable()`) then set the corresponding metatable as a metatable to the returned new
metatable.

If the given `constructor` function is not `nil`, then assign to the variable `name` the given constructor.
The given `name` might be of the form `package.className`, in which case the `className` will be local to the
specified `package`. In that case, `package` must be a valid and already loaded package.

[function] torch.factory(name)

Returns the factory function of the Torch class `name`. If the class name is invalid or if the class has no factory, then returns `nil`.

A Torch class is a class created with `torch.class()` or `torch.newmetatable()`.

A factory function is able to return a new (empty) object of its corresponding class. This is helpful for
object serialization.

[table] torch.getmetatable(string)

Given a `string`, returns a metatable corresponding to the Torch class described by `string`. Returns `nil` if the class does not exist.

A Torch class is a class created with `torch.class()` or `torch.newmetatable()`.

Example:

```
> for k, v in pairs(torch.getmetatable('torch.CharStorage')) do print(k, v) end

__index__       function: 0x1a4ba80
```

```
__typename      torch.CharStorage
write           function: 0x1a49cc0
__tostring__    function: 0x1a586e0
__newindex__    function: 0x1a4ba40
string          function: 0x1a4d860
__version       1
read            function: 0x1a4d840
copy            function: 0x1a49c80
__len__         function: 0x1a37440
fill            function: 0x1a375c0
resize          function: 0x1a37580
__index         table: 0x1a4a080
size            function: 0x1a4ba20
```

[boolean] torch.isequal(object1, object2)

If the two objects given as arguments are *Lua* tables (or *Torch7* objects), then returns `true` if and only if the
tables (or Torch objects) have the same address in memory. Returns `false` in any other cases.

A Torch class is a class created with `torch.class()` or `torch.newmetatable()`.

[string] torch.getdefaulttensortype()

Returns a string representing the default tensor type currently in use by *Torch7*.

[table] torch.getenv(function or userdata)

Returns the Lua `table` environment of the given `function` or the given `userdata`. To know more about environments, please read the documentation of `lua_setfenv()` and `lua_getfenv()`.

[number] torch.version(object)

Returns the field `lua__version` of a given object. This might be helpful to handle variations in a class over time.

[number] torch.pointer(object)

Returns a unique id (pointer) of the given `object`, which can be a *Torch7* object, a table, a thread or a function.

This is different from the `class` id returned by `torch.id()`.

torch.setdefaulttensortype([typename])

Sets the default tensor type for all the tensors allocated from this point on. Valid types are:

- `torch.ByteTensor`
- `torch.CharTensor`
- `torch.ShortTensor`
- `torch.IntTensor`
- `torch.FloatTensor`
- `torch.DoubleTensor`

torch.setenv(function or userdata, table)

Assign `table` as the Lua environment of the given `function` or the given `userdata`. To know more about environments, please read the documentation of `lua_setfenv()`
and `lua_getfenv()`.

[object] torch.setmetatable(table, classname)

Set the metatable of the given `table` to the metatable of the Torch object named `classname`. This function has to be used with a lot of care.

[table] torch.getconstructortable(string)

BUGGY
Return the constructor table of the Torch class specified by `string`.

[table] torch.totable(object)

Converts a Tensor or a Storage to a lua table. Also available as methods: `tensor:totable()` and `storage:totable()`.
Multidimensional Tensors are converted to a set of nested tables, matching the shape of the source Tensor.

```
> print(torch.totable(torch.Tensor({1, 2, 3})))
{
  1 : 1
  2 : 2
  3 : 3
}
```