Need help?

- Questions, Support, Install issues: Google groups
- Reporting bugs: torch7 nn cutorch cunn optim threads
- Hanging out with other developers and users (strictly no install issues, no large blobs of text): Gitter Chat

Torch Package Reference Manual

Torch is the main package in Torch7 where data structures for multi-dimensional tensors and mathematical operations over these are defined. Additionally, it provides many utilities for accessing files, serializing objects of arbitrary types and other useful utilities.

Torch Packages

- Tensor Library
  - Tensor defines the *all powerful* tensor object that provides multi-dimensional numerical arrays with type templating.
  - Mathematical operations that are defined for the tensor object types.
  - Storage defines a simple storage interface that controls the underlying storage for any tensor object.

- File I/O Interface Library
  - File is an abstract interface for common file operations.
  - Disk File defines operations on files stored on disk.
  - Memory File defines operations on stored in RAM.
  - Pipe File defines operations for using piped commands.
  - High-Level File operations defines higher-level serialization functions.

- Useful Utilities
  - Timer provides functionality for *measuring time*.
  - Tester is a generic tester framework.

    - CmdLine is a command line argument parsing utility.
    - Random defines a random number generator package with various distributions.
    - Finally useful utility functions are provided for easy handling of torch tensor types and class inheritance.

Useful Links

- Community packages
- Torch Blog
- Torch Slides

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.

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
```

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()
```

```
      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
}
```

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.

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.

dataload

```
local dl = require 'dataload'
```

A collection of Torch dataset loaders.
The library provides the following generic data loader classes :

- DataLoader : an abstract class inherited by the following classes;
- TensorLoader : for tensor or nested (i.e. tables of) tensor datasets;
- ImageClass : for image classification datasets stored in a flat folder structure;
- AsyncIterator : decorates a `DataLoader` for asynchronou multi-threaded iteration;
- SequenceLoader : for sequence datasets like language or time-series;
- MultiSequence : for shuffled sets of sequence datasets like shuffled sentences;
- MultiImageSequence : for suffled sets of sequences of input and target images.

The library also provides functions for downloading specific datasets
and preparing them using the above loaders :

- loadMNIST : load the MNIST handwritten digit dataset for image classification;
- loadCIFAR10 : load the CIFAR10 dataset for image classification;
- loadImageNet : load the ILSVRC2014 dataset for image classification;
- loadPTB : load the Penn Tree Bank corpus for language modeling;
- loadGBW : load the Google Billion Words corpus for language modeling;
- loadSentiment140 : load the Twitter data for sentiment analysis/classification (sad, happy).

Also, we try to provide some useful preprocessing functions :

- fitImageNormalize : normalize images by channel.

DataLoader

```
dataloader = dl.DataLoader()
```

An abstract class inherited by all `DataLoader` instances.
It wraps a data set to provide methods for accessing

`inputs` and `targets`. The data itself may be loaded from disk or memory.

[n] size()

Returns the number of samples in the `dataloader`.

[size] isize([excludedim])

Returns the `size` of `inputs`. When `excludedim` is 1 (the default),
the batch dimension is excluded from `size`.
When `inputs` is a tensor, the returned `size` is
a table of numbers. When it is a table of tensors, the returned `size`
is a table of table of numbers.

[size] tsize([excludedim])

Returns the `size` of `targets`. When `excludedim` is 1 (the default),
the batch dimension is excluded from `size`.
When `targets` is a tensor, the returned `size` is
a table of numbers. When it is a table of tensors, the returned `size`
is a table of table of numbers.

[inputs, targets] index(indices, [inputs, targets])

Returns `inputs` and `targets` containing samples indexed by `indices`.

So for example :

```
indices = torch.LongTensor{1,2,3,4,5}
inputs, targets = dataloader:index(indices)
```

would return a batch of `inputs` and `targets` containing samples 1 through 5.
When `inputs` and `targets` are provided as arguments, they are used as
memory buffers for the returned `inputs` and `targets`,
i.e. their allocated memory is reused.

[inputs, targets] sample(batchsize, [inputs, targets])

Returns `inputs` and `targets` containing `batchsize` random samples.
This method is equivalent to :

```
indices = torch.LongTensor(batchsize):random(1,dataloader:size())
inputs, targets = dataloader:index(indices)
```

[inputs, targets] sub(start, stop, [inputs, targets])

Returns `inputs` and `targets` containing `stop-start+1` samples between `start` and `stop`.
This method is equivalent to :

```
indices = torch.LongTensor():range(start, stop)
inputs, targets = dataloader:index(indices)
```

shuffle()

Internally shuffles the `inputs` and `targets`. Note that not all
subclasses support this method.

[ds1, ds2] split(ratio)

Splits the `dataloader` into two new `DataLoader` instances
where `ds1` contains the first `math.floor(ratio x dataloader:size())` samples,
and `ds2` contains the remainder.
Useful for splitting a training set into a new training set and validation set.

[iterator] subiter([batchsize, epochsize, ...])

Returns an iterator over a validation and test sets.
Each iteration returns 3 values :

- `k` : the number of samples processed so far. Each iteration returns a maximum of `batchsize` samples.
- `inputs` : a tensor (or nested table thereof) containing a maximum of `batchsize` inputs.
- `targets` : a tensor (or nested table thereof) containing targets for the commensurate inputs.

The iterator will return batches of `inputs` and `targets` of size at most `batchsize` until `epochsize` samples have been returned.

Note that the default implementation of this iterator is to call sub for each batch.
Sub-classes may over-write this behavior.

Example :

```
local dl = require 'dataload'

inputs, targets = torch.range(1,5), torch.range(1,5)
dataloader = dl.TensorLoader(inputs, targets)

local i = 0
for k, inputs, targets in dataloader:subiter(2,6) do
   i = i + 1
   print(string.format("batch %d, nsampled = %d", i, k))
   print(string.format("inputs:\n%stargets:\n%s", inputs, targets))
end
```

Output :

```
batch 1, nsampled = 2
inputs:
 1
 2
[torch.DoubleTensor of size 2]
targets:
 1
 2
[torch.DoubleTensor of size 2]

batch 2, nsampled = 4
inputs:
```

```
 3
 4
[torch.DoubleTensor of size 2]
targets:
 3
 4
[torch.DoubleTensor of size 2]

batch 3, nsampled = 5
inputs:
 5
[torch.DoubleTensor of size 1]
targets:
 5
[torch.DoubleTensor of size 1]

batch 4, nsampled = 6
inputs:
 1
[torch.DoubleTensor of size 1]
targets:
 1
[torch.DoubleTensor of size 1]
```

Note how the last two batches are of size 1 while those before are of size `batchsize = 2`.
The reason for this is that the `dataloader` only has 5 samples.
So the last batch is split between the last sample and the first.

[iterator] sampleiter([batchsize, epochsize, ...])

Returns an iterator over a training set.
Each iteration returns 3 values :

- `k` : the number of samples processed so far. Each iteration returns a maximum of `batchsize` samples.
- `inputs` : a tensor (or nested table thereof) containing a maximum of `batchsize` inputs.
- `targets` : a tensor (or nested table thereof) containing targets for the commensurate inputs.

The iterator will return batches of `inputs` and `targets` of size at most `batchsize` until `epochsize` samples have been returned.

Note that the default implementation of this iterator is to call sample for each batch.
Sub-classes may over-write this behavior.

Example :

```
local dl = require 'dataload'

inputs, targets = torch.range(1,5), torch.range(1,5)
dataloader = dl.TensorLoader(inputs, targets)

local i = 0
for k, inputs, targets in dataloader:sampleiter(2,6) do
   i = i + 1
   print(string.format("batch %d, nsampled = %d", i, k))
   print(string.format("inputs:\n%stargets:\n%s", inputs, targets))
end
```

Output :

```
batch 1, nsampled = 2
inputs:
 1
 2
[torch.DoubleTensor of size 2]
targets:
 1
 2
[torch.DoubleTensor of size 2]

batch 2, nsampled = 4
inputs:
 4
 2
[torch.DoubleTensor of size 2]
targets:
 4
 2
[torch.DoubleTensor of size 2]

batch 3, nsampled = 6
inputs:
 4
 1
```

```
[torch.DoubleTensor of size 2]
targets:
 4
 1
[torch.DoubleTensor of size 2]
```

reset()

Resets all internal counters such as those used for iterators.
Called by AsyncIterator before serializing the `DataLoader` to threads.

collectgarbage()

Collect garbage every `self.gccdelay` times this method is called.

[copy] clone()

Returns a deep `copy` clone of `self`.

TensorLoader

```
dataloader = dl.TensorLoader(inputs, targets)
```

The `TensorLoader` can be used to encapsulate tensors of `inputs` and `targets`.
As an example, consider a dummy `3 x 8 x 8` image classification dataset consisting of 1000 samples and 10 classes:

```
inputs = torch.randn(1000, 3, 8, 8)
targets = torch.LongTensor(1000):random(1,10)
dataloader = dl.TensorLoader(inputs, targets)
```

The `TensorLoader` can also be used to encapsulate nested tensors of `inputs` and `targets`.
It uses recursive functions to handle nestings of arbitrary depth. As an example, let us modify the above example to include `x,y` GPS coordinates in the `inputs` and a parallel set of classification `targets` (7 classes):

```
inputs = {torch.randn(1000, 3, 8, 8), torch.randn(1000, 2)}
targets = {torch.LongTensor(1000):random(1,10),
torch.LongTensor(1000):random(1,7)}
dataloader = dl.TensorLoader(inputs, targets)
```

ImageClass

```
dataloader = dl.ImageClass(datapath, loadsize, [samplesize,
samplefunc, sortfunc, verbose])
```

For loading an image classification data set stored in a flat folder structure :

```
(datapath)/(classdir)/(imagefile).(jpg|png|etc)
```

So directory `classdir` is expected to contain the all images belonging to that class.
All image files are indexed into an efficient `CharTensor` during initialization.
Images are only loaded into `inputs` and `targets` tensors upon calling batch sampling methods like index, sample and sub.

Note that for asynchronous loading of images (i.e. loading batches of images in different threads),
the `ImageClass` loader can be decorated with an AsyncIterator.
Images on disk can have different height, width and number of channels.

Constructor arguments are as follows :

- `datapath` : one or many paths to directories of images;
- `loadsize` : initialize size to load the images to. Example : `{3, 256, 256}` ;
- `samplesize` : consistent sample size to resize the images to. Defaults to `loadsize` ;
- `samplefunc` : `function f(self, dst, path)` used to create a sample(s) from an image path. Stores them in `CharTensor` `dst` . Strings `"sampleDefault"` (the

default), `"sampleTrain"` or `"sampleTest"` can also be provided as they refer to existing functions
- `verbose` : display verbose message (default is `true`);
- `sortfunc` : comparison operator used for sorting `classdir` to get class indices. Defaults to the `<` operator.

AsyncIterator

```
dataloader = dl.AsyncIterator(dataloader, [nthread, verbose,
serialmode])
```

This `DataLoader` subclass overwrites the `subiter` and `sampleiter` iterator methods. The implementation uses the threads package to build a pool of `nthread` worker threads. The main thread delegates the tasks of building `inputs` and `targets` tensors to the workers. The workers each have a deep copy of the decorated `dataloader`. The optional parameter serialmode can be specified as 'ascii' (default) or 'binary'. If large amounts of data need to be processed, 'binary' can prevent the dataloader from allocating too much RAM.
When a task is received from the main thread through the Queue, they call `sample` or `sub` to build the batch and return the `inputs` and `targets` to the main thread. The iteration is asynchronous as the first iteration will fill the Queue with `nthread` tasks.
Note that when `nthread > 1` the order of tensors is not deterministic.
This loader is well suited for decorating a `dl.ImageClass` instance and other such I/O and CPU bound loaders.

SequenceLoader

```
dataloader = dl.SequenceLoader(sequence, batchsize,
[bidirectional])
```

This `DataLoader` subclass can be used to encapsulate a `sequence` for training time-series or language models.

The `sequence` is a tensor where the first dimension indexes time.
Internally, the loader will split the `sequence` into `batchsize` subsequences.
Calling the `sub(start, stop, inputs, targets)` method will return
`inputs` and `targets` of size `seqlen x batchsize [x inputsize]`
where `stop - start + 1 <= seqlen`.
See RNNLM training script for an example.

The `bidirectional` argument should be set
to `true` for bidirectional models like BRNN/BLSTMs. In which case,
the returned `inputs` and `targets` will be aligned.
For example, using `batchsize = 3` and `seqlen = 5`:

```
print(inputs:t(), targets:t())
   36  1516   853    94  1376
 3193   433   553   805   521
  512   434    57  1029  1962
[torch.IntTensor of size 3x5]

   36  1516   853    94  1376
 3193   433   553   805   521
  512   434    57  1029  1962
[torch.IntTensor of size 3x5]
```

When `bidirectional` is `false` (the default), the `targets` will
be one step in the future with respect to the inputs:
For example, using `batchsize = 3` and `seqlen = 5`:

```
print(inputs:t(), targets:t())
   36  1516   853    94  1376
 3193   433   553   805   521
  512   434    57  1029  1962
[torch.IntTensor of size 3x5]

 1516   853    94  1376   719
  433   553   805   521    27
  434    57  1029  1962    49
[torch.IntTensor of size 3x5]
```

MultiSequence

```
dataloader = dl.MultiSequence(sequences, batchsize)
```

This DataLoader subclass is used by the Billion Words dataset to encapsulate unordered sentences.
The `sequences` arguments is a table or tds.Vec of tensors.
Each such tensors is a single sequence independent of the others. The tensor can be multi-dimensional as long
as the non-sequence dimension sizes are consistent from sequence to sequence.

When calling `sub(start, stop)` or `subiter(seqlen)` methods,
a column of the returned `inputs` and `targets` tensors (of size `seqlen x batchsize`) could
contain multiple sequences. For example, a character-level language model could look like:

```
target : [ ] E L L O [ ] C R E E N ...
input  : [ ] H E L L [ ] S C R E E ...
```

where `HELLO` and `SCREEN` would be two independent sequences.
Note that `[ ]` is a zero mask used to seperate independent sequences.
For most cases, the `[ ]` token is a 0.
Except for 1D `targets`, where it is a 1 (so that it works with `ClassNLLCriterion`).

MultiImageSequence

```
ds = dl.MultiImageSequence(datapath, batchsize, loadsize,
samplesize, [samplefunc, verbose])
```

This `DataLoader` is used to load datasets consisting of independent sequences of
input and target images. So basically, each independent sequence consists of
two sequences of the same size, one for inputs, one for targets.

As a concrete example, this `DataLoader` could be used to
wrap a dataset where each input is a sequence of video frames,
and its commensurate targets are binary masks.

Like the `ImageClass` loader, `MultiImageSequence` expects images to be stored on disk.
Each directory is organized as :

```
[datapath]/[seqid]/[input|target][1,2,3,...,T].jpg
```

where the `datapath` (first constructor argument) specifies the file system path to the data.
That directory is expected to contain a folder for each sequence, here represented by the `seqid` variable.
The `seqid` folder can have any name, but by default its contents are expected to contain the pattern
`input%d.jpg` and `target%d.jpg` for input and target images, respectively.
Internally, the `%d` is replaced with integers starting at `1` until no more images are found.
These patterns can be replaced after construction via the `inputpattern` and `targetpattern`.

Variable length sequences are natively supported.

Images will be only be loaded when requested.

Like the `MultiSequence` loader, the `batchsize` must be specified during construction.
Like the `ImageClass`, the `loadsize` argument specifies that size of to which the images are to be loaded initially.
These are specified as two tables in `c x h x w` format, for inputs and targets respectively (e.g. `{{3,28,28},{1,8,8}}`).

The `samplesize` specifies the returned input image size (e.g. `{3,24,24}`).
The actual sample size of the targets cannot be provided as it will be forced to be proportional to the input's load to sample size.

The `samplefunc` specifies the function to use for sampling input and target images.
The default value of `sampleDefault` simply resizes the images to the given input `samplesize` and the proportional target sample size.
When `sampleTrain` is provided, a random location will be chosen for each sampled sequence.

When calling `sub(start, stop)` the returned `input` and `target` are tensors of size `seqlen x batchsize x samplesize`. Since variable length sequences are natively supported, the returned `inputs` and `targets` will be separated by mask tokens (here represented by `[ ]`):

```
[ ] target11, target12, target13, ..., target1T [ ] target21, ...
[ ] input11,  input12,  input13,  ..., input1T  [ ] input21, ...
```

The mask tokens `[ ]` represent images with nothing but zeros.

For large datasets use Lua5.2 instead of LuaJIT to avoid memory errors (see torch.ch).

The following are attributes that can be set to `true` to modify the behavior of the loader:

- `cropeverystep` : samples a random uniform crop location every time-step (instead of once per sequence)
- `varyloadsize` : random-uniformly samples a `loadsize` between samplesize and loadsize (this effectively scales the cropped location)
- `scaleeverystep` : varies `loadsize` every step instead of once per sequence
- `randseq` : each new sequence is chosen random uniformly

loadMNIST

```
train, valid, test = dl.loadMNIST([datapath, validratio, scale,
srcurl])
```

Returns the training, validation and testing sets as 3 `TensorLoader` instances.
Each such loader encapsulates a part of the MNIST dataset which is
located in `datapath` (defaults to `dl.DATA_PATH/mnist`).
The `validratio` argument, a number between 0 and 1,
specifies the ratio of the 60000 training samples
that will be allocated to the validation set.
The `scale` argument specifies range within which pixel values will be scaled (defaults to `{0,1}`).
The `srcurl` specifies the URL from where the raw data can be downloaded from
if not located on disk.

loadCIFAR10

```
train, valid, test = dl.loadCIFAR10([datapath, validratio, scale,
srcurl])
```

Returns the training, validation and testing sets as 3 `TensorLoader` instances.
Each such loader encapsulates a part of the CIFAR10 dataset which is
located in `datapath` (defaults to `dl.DATA_PATH/cifar-10-batches-t7`).
The `validratio` argument, a number between 0 and 1,
specifies the ratio of the 50000 training samples

that will be allocated to the validation set.
The `scale` argument specifies range within which pixel values will be scaled (defaults to `{0,1}`).
The `srcurl` specifies the URL from where the raw data can be downloaded from
if not located on disk.

loadPTB

```
train, valid, test = dl.loadPTB(batchsize, [datapath, srcurl])
```

Returns the training, validation and testing sets as 3 `SequenceLoader` instance
Each such loader encapsulates a part of the Penn Tree Bank dataset which is
located in `datapath` (defaults to `dl.DATA_PATH/PennTreeBank`).
If the files aren't found in the `datapath`, they will be automatically downloaded
from the `srcurl` URL.
The `batchsize` specifies the number of samples that will be returned when
iterating through the dataset. If specified as a table, its elements
specify the `batchsize` of commensurate `train`, `valid` and `test` tables.
We recommend a `batchsize` of 1 for evaluation sets (e.g. `{50,1,1}`).

See RNNLM training script for an example.

loadImageNet

Ref.: A. http://image-net.org/challenges/LSVRC/2014/download-images-5jj5.php

```
train, valid = dl.loadImageNet(datapath, [nthread, loadsize,
samplesize, verbose])
```

Returns the training and validation sets of the Large Scale Visual Recognition Challenge 2014
(ILSVRC2014)
image classification dataset (commonly known as ImageNet).
The dataset hasn't changed from 2012-2014.

The returned `train` and `valid` loaders do not read all images into memory when first

loaded.
Each dataset is implemented using an ImageClass loader decorated by an AsyncIterator.

The `datapath` should point to a directory containing the outputs of the `downloadimagenet.lua` and `harmonizeimagenet.lua` scripts (see bellow).

Requirements

Due to its size, the data first needs to be prepared offline.
Use downloadimagenet.lua
to download and extract the data :

```
th downloadimagenet.lua --savePath '/path/to/diskspace/ImageNet'
```

The entire process requires about 360 GB of disk space to complete the download and extraction process.
This can be reduced to about 150 GB if the training set is downloaded and extracted first, and all the `.tar` files are manually deleted. Repeat for the validation set, devkit and metadata.
If you still don't have enough space in one partition, you can divide the data among different partitions.
We recommend a good internet connection (>60Mbs download) and a Solid-State Drives (SSD).

Use harmonizeimagenet.lua
to harmonize the train and validation sets:

```
th harmonizeimagenet.lua --dataPath /path/to/diskspace/ImageNet --
progress --forReal
```

Each set will then contain a directory of images for each class with name `class[id]` where `[id]` is a class index, between 1 and 1000, used for the ILVRC2014 competition.

Then we need to install graphicsmagick :

```
luarocks install graphicsmagick
```

Inference

As in the famous (Krizhevsky et al. 2012)
paper, the ImageNet training dataset samples images cropped from random
224x224 patches from the images resizes so that the smallest dimension has
size 256. As for the validation set, ten 224x224 patches are cropped per image,
i.e. center, four corners and their horizontal flips, and their predictions are averaged.

loadGBW

```
train, valid, test = dl.loadGBW(batchsize, [trainfile, datapath,
srcurl, verbose])
```

Loads the Google Billion Words corpus as MultiSequence loaders.
The preprocessing specified in
Google Billion Words language modeling benchmark
was applied to `training-monolingual.tokenized/news.20??.en.shuffled.tokenized` to generate the different subsets.
These subsets are automatically downloaded when not found on disk.
The task consists in predicting the next word given the previous ones.
The corpus contains approximately 30 million sentences of an average length of about 25 words.
In total, there are about 800 thousand (unique) words in the vocabulary, which makes it a very memory intensive problem.

loadSentiment140

```
train, valid, test = dl.loadSentiment140([datapath, minfreq,
seqlen, validratio, srcurl, progress])

Load & processing training data.
Number of tweets: 1600000
Vocabulary size: 155723
```

```
Number of occurences replaced with <OOV> token: 750575
Tweet corpus size (in number of tokens): 20061241
trainset set processed in 28.306740999222s
```

Load the Sentiment140 dataset.
This dataset can be used for sentiment analysis for microblogging websites like Twitter.
The task is to predict the sentiment of a tweet.
The input is a sequence of tokenized words with a default maximum sequence length of 50 (i.e. `seqlen=50`).
Targets can be one of three classes that map to the sentiment of the tweet: 1 = negative, 2 = neutral, 3 = positive. The neutral tweets are not present in the training data hence we ignore them from all (train, valid & test) datasets. This results in a 2-class (1=Negative, 2=Positive) dataset.
Tweets are tokenized using the `twitter/twokenize.py` script.
By default, only words with at least 3 occurrences (i.e. `minfreq=3`) in the training set are kept.
The dataset is automatically downloaded from `srcurl`, tokenized and parsed into a tensor the first time the loader is used.
The returned training, validation and test sets are encapsulated using the TensorLoader.
The input is padded with zeros before the tweet when it is shorted than `seqlen`.

The above is only printed when `progress=true` (the default) the first time the loader is invoked.
The processed data is subsequently cached to speedup future loadings.
To overwrite any cached data use `dl.overwrite=true`.

fitImageNormalize

```
ppf = dl.fitImageNormalize(trainset, [nsample, cachepath, verbose])
```

Returns a `ppf` preprocessing function that can be used to in-place normalize a batch of images (`inputs`)
channel-wise:

```
ppf(inputs)
```

The `trainset` argument is a DataLoader instance
containing image `inputs`. The mean and standard deviation will be measured
on `nsample` images (default 10000). When `cachepath` is provided, the

mean and standard deviation are saved for the next function call.

Overview

Each module of a network is composed of Modules and there are several sub-classes of `Module` available: container classes like Sequential, Parallel and Concat , which can contain simple layers like Linear, Mean, Max and Reshape, as well as convolutional layers, and transfer functions like Tanh.

Loss functions are implemented as sub-classes of Criterion. They are helpful to train neural network on classical tasks. Common criterions are the Mean Squared Error criterion implemented in MSECriterion and the cross-entropy criterion implemented in ClassNLLCriterion.

Finally, the StochasticGradient class provides a high level way to train the neural network of choice, even though it is easy with a simple for loop to train a neural network yourself.

Detailed Overview

This section provides a detailed overview of the neural network package. First the omnipresent Module is examined, followed by some examples for combining modules together. The last part explores facilities for training a neural network, and finally some caveats while training networks with shared parameters.

Module

A neural network is called a Module (or simply *module* in this documentation) in Torch. `Module` is an abstract class which defines four main methods:

- forward(input) which computes the output of the module given the `input` Tensor.
- backward(input, gradOutput) which computes the gradients of the module with respect

to its own parameters, and its own inputs.
- zeroGradParameters() which zeroes the gradient with respect to the parameters of the module.
- updateParameters(learningRate) which updates the parameters after one has computed the gradients with `backward()`

It also declares two members:

- output which is the output returned by `forward()`.
- gradInput which contains the gradients with respect to the input of the module, computed in a `backward()`.

Two other perhaps less used but handy methods are also defined:

- share(mlp,s1,s2,...,sn) which makes this module share the parameters s1,..sn of the module `mlp`. This is useful if you want to have modules that share the same weights.
- clone(...) which produces a deep copy of (i.e. not just a pointer to) this Module, including the current state of its parameters (if any).

Some important remarks:

- `output` contains only valid values after a forward(input).
- `gradInput` contains only valid values after a backward(input, gradOutput).
- backward(input, gradOutput) uses certain computations obtained during forward(input). You *must* call `forward()` before calling a `backward()`, on the *same* `input`, or your gradients are going to be incorrect!

Plug and play

Building a simple neural network can be achieved by constructing an available layer.
A linear neural network (perceptron!) is built only in one line:

```
mlp = nn.Linear(10,1) -- perceptron with 10 inputs
```

More complex neural networks are easily built using container classes Sequential and Concat. `Sequential` plugs layer in a feed-forward fully connected manner. `Concat` concatenates in one layer several modules: they take the same inputs, and their output is concatenated.

Creating a one hidden-layer multi-layer perceptron is thus just as easy as:

```
mlp = nn.Sequential()
mlp:add( nn.Linear(10, 25) ) -- 10 input, 25 hidden units
mlp:add( nn.Tanh() ) -- some hyperbolic tangent transfer function
mlp:add( nn.Linear(25, 1) ) -- 1 output
```

Of course, `Sequential` and `Concat` can contains other `Sequential` or `Concat`, allowing you to try the craziest neural networks you ever dreamt of!

Training a neural network

Once you built your neural network, you have to choose a particular Criterion to train it. A criterion is a class which describes the cost to be minimized during training.

You can then train the neural network by using the StochasticGradient class.

```
criterion = nn.MSECriterion() -- Mean Squared Error criterion
trainer = nn.StochasticGradient(mlp, criterion)
trainer:train(dataset) -- train using some examples
```

StochasticGradient expect as a `dataset` an object which implements the operator `dataset[index]` and implements the method `dataset:size()`. The `size()` methods returns the number of examples and `dataset[i]` has to return the i-th example.

An `example` has to be an object which implements the operator `example[field]`, where `field` might take the value `1` (input features) or `2` (corresponding label which will be given to the criterion). The input is usually a Tensor (except if you use special kind of gradient modules, like table layers). The label type depends on the criterion. For example, the MSECriterion expect a Tensor, but the ClassNLLCriterion except a integer number (the class).

Such a dataset is easily constructed by using Lua tables, but it could any `C` object for example, as long as required operators/methods are implemented. See an example.

`StochasticGradient` being written in `Lua`, it is extremely easy to cut-and-paste it and create a variant to it adapted to your needs (if the constraints of `StochasticGradient` do not satisfy you).

Low Level Training

If you want to program the `StochasticGradient` by hand, you essentially need to control the use of forwards and backwards through the network yourself. For example, here is the code fragment one would need to make a gradient step given an input `x`, a desired output `y`, a network `mlp` and a given criterion `criterion` and learning rate `learningRate`:

```
function gradUpdate(mlp, x, y, criterion, learningRate)
  local pred = mlp:forward(x)
  local err = criterion:forward(pred, y)
  local gradCriterion = criterion:backward(pred, y)
  mlp:zeroGradParameters()
  mlp:backward(x, gradCriterion)
  mlp:updateParameters(learningRate)
end
```

For example, if you wish to use your own criterion you can simply replace `gradCriterion` with the gradient vector of your criterion of choice.

A Note on Sharing Parameters

By using `:share(...)` and the Container Modules, one can easily create very complex architectures. In order to make sure that the network is going to train properly, one needs to pay attention to the way the sharing is applied, because it might depend on the optimization procedure.

- If you are using an optimization algorithm that iterates over the modules of your network (by calling `:updateParameters` for example), only the parameters of the network should be shared.
- If you use the flattened parameter tensor to optimize the network, obtained by calling `:getParameters`, for example for the package `optim`, then you need to share both the parameters and the gradParameters.

Here is an example for the first case:

```
-- our optimization procedure will iterate over the modules, so
only share
-- the parameters
mlp = nn.Sequential()
linear = nn.Linear(2,2)
linear_clone = linear:clone('weight','bias') -- clone sharing the
parameters
mlp:add(linear)
mlp:add(linear_clone)
function gradUpdate(mlp, x, y, criterion, learningRate)
  local pred = mlp:forward(x)
  local err = criterion:forward(pred, y)
  local gradCriterion = criterion:backward(pred, y)
  mlp:zeroGradParameters()
  mlp:backward(x, gradCriterion)
  mlp:updateParameters(learningRate)
end
```

And for the second case:

```
-- our optimization procedure will use all the parameters at once,
because
-- it requires the flattened parameters and gradParameters Tensors.
Thus,
-- we need to share both the parameters and the gradParameters
mlp = nn.Sequential()
linear = nn.Linear(2,2)
-- need to share the parameters and the gradParameters as well
linear_clone =
linear:clone('weight','bias','gradWeight','gradBias')
mlp:add(linear)
mlp:add(linear_clone)
params, gradParams = mlp:getParameters()
function gradUpdate(mlp, x, y, criterion, learningRate, params,
gradParams)
  local pred = mlp:forward(x)
  local err = criterion:forward(pred, y)
  local gradCriterion = criterion:backward(pred, y)
  mlp:zeroGradParameters()
  mlp:backward(x, gradCriterion)
  -- adds the gradients to all the parameters at once
  params:add(-learningRate, gradParams)
end
```

Module

`Module` is an abstract class which defines fundamental methods necessary for a training a neural network. Modules are serializable.

Modules contain two states variables: output and gradInput.

[output] forward(input)

Takes an `input` object, and computes the corresponding `output` of the module. In general `input` and `output` are Tensors. However, some special sub-classes like table layers might expect something else. Please, refer to each module specification for further information.

After a `forward()`, the output state variable should have been updated to the new value.

It is not advised to override this function. Instead, one should implement updateOutput(input) function. The forward module in the abstract parent class Module will call `updateOutput(input)`.

[gradInput] backward(input, gradOutput)

Performs a *backpropagation step* through the module, with respect to the given `input`. In general this method makes the assumption forward(input) has been called before, *with the same input*. This is necessary for optimization reasons. If you do not respect this rule, `backward()` will compute incorrect gradients.

In general `input` and `gradOutput` and `gradInput` are Tensors. However, some special sub-classes like table layers might expect something else. Please, refer to each module specification for further information.

A *backpropagation step* consist in computing two kind of gradients

at `input` given `gradOutput` (gradients with respect to the output of the module). This function simply performs this task using two function calls:

- A function call to updateGradInput(input, gradOutput).
- A function call to accGradParameters(input,gradOutput,scale).

It is not advised to override this function call in custom classes. It is better to override updateGradInput(input, gradOutput) and accGradParameters(input, gradOutput,scale) functions.

updateOutput(input)

Computes the output using the current parameter set of the class and input. This function returns the result which is stored in the output field.

updateGradInput(input, gradOutput)

Computing the gradient of the module with respect to its own input. This is returned in `gradInput`. Also, the gradInput state variable is updated accordingly.

accGradParameters(input, gradOutput, scale)

Computing the gradient of the module with respect to its own parameters. Many modules do not perform this step as they do not have any parameters. The state variable name for the parameters is module dependent. The module is expected to *accumulate* the gradients with respect to the parameters in some variable.

`scale` is a scale factor that is multiplied with the gradParameters before being accumulated.

Zeroing this accumulation is achieved with zeroGradParameters() and updating

the parameters according to this accumulation is done with updateParameters().

zeroGradParameters()

If the module has parameters, this will zero the accumulation of the gradients with respect to these parameters, accumulated through accGradParameters(input, gradOutput,scale) calls. Otherwise, it does nothing.

updateParameters(learningRate)

If the module has parameters, this will update these parameters, according to the accumulation of the gradients with respect to these parameters, accumulated through backward() calls.

The update is basically:

```
parameters = parameters - learningRate * gradients_wrt_parameters
```

If the module does not have parameters, it does nothing.

accUpdateGradParameters(input, gradOutput, learningRate)

This is a convenience module that performs two functions at once. Calculates and accumulates the gradients with respect to the weights after multiplying with negative of the learning rate `learningRate`. Performing these two operations at once is more performance efficient and it might be advantageous in certain situations.

Keep in mind that, this function uses a simple trick to achieve its goal and it might not be valid for a custom module.

Also note that compared to accGradParameters(), the gradients are not retained for future use.

```
function Module:accUpdateGradParameters(input, gradOutput, lr)
   local gradWeight = self.gradWeight
   local gradBias = self.gradBias
   self.gradWeight = self.weight
   self.gradBias = self.bias
   self:accGradParameters(input, gradOutput, -lr)
   self.gradWeight = gradWeight
   self.gradBias = gradBias
end
```

As it can be seen, the gradients are accumulated directly into weights. This assumption may not be true for a module that computes a nonlinear operation.

share(mlp,s1,s2,...,sn)

This function modifies the parameters of the module named `s1`,.. `sn` (if they exist) so that they are shared with (pointers to) the parameters with the same names in the given module `mlp`.

The parameters have to be Tensors. This function is typically used if you want to have modules that share the same weights or biases.

Note that this function if called on a Container module will share the same parameters for all the contained modules as well.

Example:

```
-- make an mlp
mlp1=nn.Sequential();
mlp1:add(nn.Linear(100,10));

-- make a second mlp
mlp2=nn.Sequential();
mlp2:add(nn.Linear(100,10));

-- the second mlp shares the bias of the first
mlp2:share(mlp1,'bias');
```

```
-- we change the bias of the first
mlp1:get(1).bias[1]=99;

-- and see that the second one's bias has also changed..
print(mlp2:get(1).bias[1])
```

clone(mlp,...)

Creates a deep copy of (i.e. not just a pointer to) the module, including the current state of its parameters (e.g. weight, biases etc., if any).

If arguments are provided to the `clone(...)` function it also calls share(...) with those arguments on the cloned module after creating it, hence making a deep copy of this module with some shared parameters.

Example:

```
-- make an mlp
mlp1=nn.Sequential();
mlp1:add(nn.Linear(100,10));

-- make a copy that shares the weights and biases
mlp2=mlp1:clone('weight','bias');

-- we change the bias of the first mlp
mlp1:get(1).bias[1]=99;

-- and see that the second one's bias has also changed..
print(mlp2:get(1).bias[1])
```

type(type[, tensorCache])

This function converts all the parameters of a module to the given `type`. The `type` can be one of the types defined for torch.Tensor.

If tensors (or their storages) are shared between multiple modules in a

network, this sharing will be preserved after type is called.

To preserve sharing between multiple modules and/or tensors, use `nn.utils.recursiveType`:

```
-- make an mlp
mlp1=nn.Sequential();
mlp1:add(nn.Linear(100,10));

-- make a second mlp
mlp2=nn.Sequential();
mlp2:add(nn.Linear(100,10));

-- the second mlp shares the bias of the first
mlp2:share(mlp1,'bias');

-- mlp1 and mlp2 will be converted to float, and will share bias
-- note: tensors can be provided as inputs as well as modules
nn.utils.recursiveType({mlp1, mlp2}, 'torch.FloatTensor')
```

float([tensorCache])

Convenience method for calling module:type('torch.FloatTensor'[, tensorCache])

double([tensorCache])

Convenience method for calling module:type('torch.DoubleTensor'[, tensorCache])

cuda([tensorCache])

Convenience method for calling module:type('torch.CudaTensor'[, tensorCache])

State Variables

These state variables are useful objects if one wants to check the guts of

a `Module`. The object pointer is *never* supposed to change. However, its contents (including its size if it is a Tensor) are supposed to change.

In general state variables are
Tensors.
However, some special sub-classes
like table layers contain something else. Please,
refer to each module specification for further information.

output

This contains the output of the module, computed with the last call of forward(input).

gradInput

This contains the gradients with respect to the inputs of the module, computed with the last call of
updateGradInput(input, gradOutput).

Parameters and gradients w.r.t parameters

Some modules contain parameters (the ones that we actually want to train!). The name of these parameters, and gradients w.r.t these parameters are module dependent.

[{weights}, {gradWeights}] parameters()

This function should returns two tables. One for the learnable parameters `{weights}` and another for the gradients of the energy wrt to the learnable parameters `{gradWeights}`.

Custom modules should override this function if they use learnable parameters that are stored in tensors.

[flatParameters, flatGradParameters] getParameters()

This function returns two tensors. One for the flattened learnable parameters `flatParameters` and another for the gradients of the energy wrt to the learnable parameters `flatGradParameters`.

Custom modules should not override this function. They should instead override parameters(…) which is, in turn, called by the present function.

This function will go over all the weights and gradWeights and make them view into a single tensor (one for weights and one for gradWeights). Since the storage of every weight and gradWeight is changed, this function should be called only once on a given network.

training()

This sets the mode of the Module (or sub-modules) to `train=true`. This is useful for modules like Dropout or BatchNormalization that have a different behaviour during training vs evaluation.

evaluate()

This sets the mode of the Module (or sub-modules) to `train=false`. This is useful for modules like Dropout or BatchNormalization that have a different behaviour during training vs evaluation.

findModules(typename)

Find all instances of modules in the network of a certain `typename`. It returns a flattened list of the matching nodes, as well as a flattened list of the container modules for each matching node.

Modules that do not have a parent container (ie, a top level nn.Sequential for instance) will return their `self` as the container.

This function is very helpful for navigating complicated nested networks. For example, a didactic example might be; if you wanted to print the output size of all `nn.SpatialConvolution` instances:

```
-- Construct a multi-resolution convolution network (with 2
resolutions):
```

```
model = nn.ParallelTable()
conv_bank1 = nn.Sequential()
conv_bank1:add(nn.SpatialConvolution(3,16,5,5))
conv_bank1:add(nn.Threshold())
model:add(conv_bank1)
conv_bank2 = nn.Sequential()
conv_bank2:add(nn.SpatialConvolution(3,16,5,5))
conv_bank2:add(nn.Threshold())
model:add(conv_bank2)
-- FPROP a multi-resolution sample
input = {torch.rand(3,128,128), torch.rand(3,64,64)}
model:forward(input)
-- Print the size of the Threshold outputs
conv_nodes = model:findModules('nn.SpatialConvolution')
for i = 1, #conv_nodes do
  print(conv_nodes[i].output:size())
end
```

Another use might be to replace all nodes of a certain `typename` with another. For instance, if we wanted to replace all `nn.Threshold` with `nn.Tanh` in the model above:

```
threshold_nodes, container_nodes =
model:findModules('nn.Threshold')
for i = 1, #threshold_nodes do
  -- Search the container for the current threshold node
  for j = 1, #(container_nodes[i].modules) do
    if container_nodes[i].modules[j] == threshold_nodes[i] then
      -- Replace with a new instance
      container_nodes[i].modules[j] = nn.Tanh()
    end
  end
end
```

listModules()

List all Modules instances in a network. Returns a flattened list of modules, including container modules (which will be listed first), self, and any other component modules.

For example :

```
mlp = nn.Sequential()
mlp:add(nn.Linear(10,20))
mlp:add(nn.Tanh())
mlp2 = nn.Parallel()
mlp2:add(mlp)
mlp2:add(nn.ReLU())
for i,module in ipairs(mlp2:listModules()) do
   print(module)
end
```

Which will result in the following output :

```
nn.Parallel {
  input
    |`-> (1): nn.Sequential {
    |      [input -> (1) -> (2) -> output]
    |      (1): nn.Linear(10 -> 20)
    |      (2): nn.Tanh
    |    }
    |`-> (2): nn.ReLU
     ... -> output
}
nn.Sequential {
  [input -> (1) -> (2) -> output]
  (1): nn.Linear(10 -> 20)
  (2): nn.Tanh
}
nn.Linear(10 -> 20)
nn.Tanh
nn.ReLU
```

clearState()

Clears intermediate module states as `output`, `gradInput` and others.
Useful when serializing networks and running low on memory. Internally calls `set()` on tensors so it does not break buffer sharing.

apply(function)

Calls provided function on itself and all child modules. This function takes module to operate on as a first argument:

```
model:apply(function(module)
   module.train = true
end)
```

In the example above `train` will be set to to `true` in all modules of `model`. This is how `training()` and `evaluate()` functions implemented.

replace(function)

Similar to apply takes a function which applied to all modules of a model, but uses return value to replace the module. Can be used to replace all modules of one type to another or remove certain modules.

For example, can be used to remove `nn.Dropout` layers by replacing them with `nn.Identity`:

```
model:replace(function(module)
   if torch.typename(module) == 'nn.Dropout' then
      return nn.Identity()
   else
      return module
   end
end)
```

Containers

Complex neural networks are easily built using container classes:

- Container : abstract class inherited by containers ;
    - Sequential : plugs layers in a feed-forward fully connected manner ;
    - Parallel : applies its `ith` child module to the `ith` slice of the input Tensor ;
    - Concat : concatenates in one layer several modules along dimension `dim` ;
        - DepthConcat : like Concat, but adds zero-padding when non-`dim` sizes don't match;
    - Bottle : allows any dimensionality input be forwarded through a module ;

See also the Table Containers for manipulating tables of Tensors.

Container

This is an abstract Module class which declares methods defined in all containers.
It reimplements many of the Module methods such that calls are propagated to the
contained modules. For example, a call to zeroGradParameters
will be propagated to all contained modules.

add(module)

Adds the given `module` to the container. The order is important

get(index)

Returns the contained modules at index `index` .

size()

Returns the number of contained modules.

Sequential

Sequential provides a means to plug layers together
in a feed-forward fully connected manner.

E.g.
creating a one hidden-layer multi-layer perceptron is thus just as easy as:

```
mlp = nn.Sequential()
mlp:add(nn.Linear(10, 25)) -- Linear module (10 inputs, 25 hidden
units)
mlp:add(nn.Tanh())         -- apply hyperbolic tangent transfer
function on each hidden units
mlp:add(nn.Linear(25, 1))  -- Linear module (25 inputs, 1 output)

> mlp
nn.Sequential {
  [input -> (1) -> (2) -> (3) -> output]
  (1): nn.Linear(10 -> 25)
  (2): nn.Tanh
  (3): nn.Linear(25 -> 1)
}

> print(mlp:forward(torch.randn(10)))
-0.1815
[torch.Tensor of dimension 1]
```

remove([index])

Remove the module at the given `index`. If `index` is not specified, remove the last layer.

```
model = nn.Sequential()
model:add(nn.Linear(10, 20))
model:add(nn.Linear(20, 20))
model:add(nn.Linear(20, 30))
```

```
model:remove(2)
> model
nn.Sequential {
  [input -> (1) -> (2) -> output]
  (1): nn.Linear(10 -> 20)
  (2): nn.Linear(20 -> 30)
}
```

insert(module, [index])

Inserts the given `module` at the given `index`. If `index` is not specified, the incremented length of the sequence is used and so this is equivalent to use `add(module)`.

```
model = nn.Sequential()
model:add(nn.Linear(10, 20))
model:add(nn.Linear(20, 30))
model:insert(nn.Linear(20, 20), 2)
> model
nn.Sequential {
  [input -> (1) -> (2) -> (3) -> output]
  (1): nn.Linear(10 -> 20)
  (2): nn.Linear(20 -> 20)      -- The inserted layer
  (3): nn.Linear(20 -> 30)
}
```

Parallel

`module` = `Parallel(inputDimension,outputDimension)`

Creates a container module that applies its `ith` child module to the `ith` slice of the input Tensor by using select
on dimension `inputDimension`. It concatenates the results of its contained modules together along dimension `outputDimension`.

Example:

```
mlp = nn.Parallel(2,1);   -- Parallel container will associate a
```

```
module to each slice of dimension 2
                            -- (column space), and concatenate the
outputs over the 1st dimension.

mlp:add(nn.Linear(10,3)); -- Linear module (input 10, output 3),
applied on 1st slice of dimension 2
mlp:add(nn.Linear(10,2))  -- Linear module (input 10, output 2),
applied on 2nd slice of dimension 2

                                  -- After going through the Linear
module the outputs are
                                  -- concatenated along the unique
dimension, to form 1D Tensor
> mlp:forward(torch.randn(10,2)) -- of size 5.
-0.5300
-1.1015
 0.7764
 0.2819
-0.6026
[torch.Tensor of dimension 5]
```

A more complicated example:

```
mlp = nn.Sequential();
c = nn.Parallel(1,2)     -- Parallel container will associate a
module to each slice of dimension 1
                         -- (row space), and concatenate the
outputs over the 2nd dimension.

for i=1,10 do            -- Add 10 Linear+Reshape modules in
parallel (input = 3, output = 2x1)
 local t=nn.Sequential()
 t:add(nn.Linear(3,2))   -- Linear module (input = 3, output = 2)
 t:add(nn.Reshape(2,1))  -- Reshape 1D Tensor of size 2 to 2D
Tensor of size 2x1
 c:add(t)
end

mlp:add(c)               -- Add the Parallel container in the
Sequential container

pred = mlp:forward(torch.randn(10,3)) -- 2D Tensor of size 10x3
goes through the Sequential container
```

```
                                        -- which contains a Parallel
container of 10 Linear+Reshape.
                                        -- Each Linear+Reshape module
receives a slice of dimension 1
                                        -- which corresponds to a 1D
Tensor of size 3.
                                        -- Eventually all the
Linear+Reshape modules' outputs of size 2x1
                                        -- are concatenated alond the
2nd dimension (column space)
                                        -- to form pred, a 2D Tensor
of size 2x10.

> pred
-0.7987 -0.4677 -0.1602 -0.8060  1.1337 -0.4781  0.1990  0.2665
-0.1364  0.8109
-0.2135 -0.3815  0.3964 -0.4078  0.0516 -0.5029 -0.9783 -0.5826
0.4474  0.6092
[torch.DoubleTensor of size 2x10]


for i = 1, 10000 do     -- Train for a few iterations
 x = torch.randn(10,3);
 y = torch.ones(2,10);
 pred = mlp:forward(x)

 criterion = nn.MSECriterion()
 local err = criterion:forward(pred,y)
 local gradCriterion = criterion:backward(pred,y);
 mlp:zeroGradParameters();
 mlp:backward(x, gradCriterion);
 mlp:updateParameters(0.01);
 print(err)
end
```

Concat

```
module = nn.Concat(dim)
```

Concat concatenates the output of one layer of "parallel" modules along the provided dimension `dim`: they take the same inputs, and their output is concatenated.

```
mlp = nn.Concat(1);
mlp:add(nn.Linear(5,3))
mlp:add(nn.Linear(5,7))

> print(mlp:forward(torch.randn(5)))
 0.7486
 0.1349
 0.7924
-0.0371
-0.4794
 0.3044
-0.0835
-0.7928
 0.7856
-0.1815
[torch.Tensor of dimension 10]
```

DepthConcat

```
module = nn.DepthConcat(dim)
```

DepthConcat concatenates the output of one layer of "parallel" modules along the provided dimension `dim`: they take the same inputs, and their output is concatenated. For dimensions other than `dim` having different sizes, the smaller tensors are copied in the center of the output tensor, effectively padding the borders with zeros.

The module is particularly useful for concatenating the output of Convolutions along the depth dimension (i.e. `nOutputFrame`).
This is used to implement the *DepthConcat* layer of the Going deeper with convolutions article.
The normal Concat Module can't be used since the spatial dimensions (height and width) of the output Tensors requiring concatenation may have different values. To deal with this, the output uses the largest

spatial dimensions and adds zero-padding around the smaller Tensors.

```
inputSize = 3
outputSize = 2
input = torch.randn(inputSize,7,7)

mlp=nn.DepthConcat(1);
mlp:add(nn.SpatialConvolutionMM(inputSize, outputSize, 1, 1))
mlp:add(nn.SpatialConvolutionMM(inputSize, outputSize, 3, 3))
mlp:add(nn.SpatialConvolutionMM(inputSize, outputSize, 4, 4))

> print(mlp:forward(input))
(1,.,.) =
 -0.2874  0.6255  1.1122  0.4768  0.9863 -0.2201 -0.1516
  0.2779  0.9295  1.1944  0.4457  1.1470  0.9693  0.1654
 -0.5769 -0.4730  0.3283  0.6729  1.3574 -0.6610  0.0265
  0.3767  1.0300  1.6927  0.4422  0.5837  1.5277  1.1686
  0.8843 -0.7698  0.0539 -0.3547  0.6904 -0.6842  0.2653
  0.4147  0.5062  0.6251  0.4374  0.3252  0.3478  0.0046
  0.7845 -0.0902  0.3499  0.0342  1.0706 -0.0605  0.5525

(2,.,.) =
 -0.7351 -0.9327 -0.3092 -1.3395 -0.4596 -0.6377 -0.5097
 -0.2406 -0.2617 -0.3400 -0.4339 -0.3648  0.1539 -0.2961
 -0.7124 -1.2228 -0.2632  0.1690  0.4836 -0.9469 -0.7003
 -0.0221  0.1067  0.6975 -0.4221 -0.3121  0.4822  0.6617
  0.2043 -0.9928 -0.9500 -1.6107  0.1409 -1.3548 -0.5212
 -0.3086 -0.0298 -0.2031  0.1026 -0.5785 -0.3275 -0.1630
  0.0596 -0.6097  0.1443 -0.8603 -0.2774 -0.4506 -0.5367

(3,.,.) =
  0.0000  0.0000  0.0000  0.0000  0.0000  0.0000  0.0000
  0.0000 -0.7326  0.3544  0.1821  0.4796  1.0164  0.0000
  0.0000 -0.9195 -0.0567 -0.1947  0.0169  0.1924  0.0000
  0.0000  0.2596  0.6766  0.0939  0.5677  0.6359  0.0000
  0.0000 -0.2981 -1.2165 -0.0224 -1.1001  0.0008  0.0000
  0.0000 -0.1911  0.2912  0.5092  0.2955  0.7171  0.0000
  0.0000  0.0000  0.0000  0.0000  0.0000  0.0000  0.0000

(4,.,.) =
  0.0000  0.0000  0.0000  0.0000  0.0000  0.0000  0.0000
  0.0000 -0.8263  0.3646  0.6750  0.2062  0.2785  0.0000
  0.0000 -0.7572  0.0432 -0.0821  0.4871  1.9506  0.0000
  0.0000 -0.4609  0.4362  0.5091  0.8901 -0.6954  0.0000
  0.0000  0.6049 -0.1501 -0.4602 -0.6514  0.5439  0.0000
```

```
    0.0000  0.2570  0.4694 -0.1262  0.5602  0.0821  0.0000
    0.0000  0.0000  0.0000  0.0000  0.0000  0.0000  0.0000

(5,.,.) =
    0.0000  0.0000  0.0000  0.0000  0.0000  0.0000  0.0000
    0.0000  0.3158  0.4389 -0.0485 -0.2179  0.0000  0.0000
    0.0000  0.1966  0.6185 -0.9563 -0.3365  0.0000  0.0000
    0.0000 -0.2892 -0.9266 -0.0172 -0.3122  0.0000  0.0000
    0.0000 -0.6269  0.5349 -0.2520 -0.2187  0.0000  0.0000
    0.0000  0.0000  0.0000  0.0000  0.0000  0.0000  0.0000
    0.0000  0.0000  0.0000  0.0000  0.0000  0.0000  0.0000

(6,.,.) =
    0.0000  0.0000  0.0000  0.0000  0.0000  0.0000  0.0000
    0.0000  1.1148  0.2324 -0.1093  0.5024  0.0000  0.0000
    0.0000 -0.2624 -0.5863  0.3444  0.3506  0.0000  0.0000
    0.0000  0.1486  0.8413  0.6229 -0.0130  0.0000  0.0000
    0.0000  0.8446  0.3801 -0.2611  0.8140  0.0000  0.0000
    0.0000  0.0000  0.0000  0.0000  0.0000  0.0000  0.0000
    0.0000  0.0000  0.0000  0.0000  0.0000  0.0000  0.0000
[torch.DoubleTensor of dimension 6x7x7]
```

Note how the last 2 of 6 filter maps have 1 column of zero-padding
on the left and top, as well as 2 on the right and bottom.
This is inevitable when the component
module output tensors non-`dim` sizes aren't all odd or even.
Such that in order to keep the mappings aligned, one need
only ensure that these be all odd (or even).

Bottle

```
module = nn.Bottle(module, [nInputDim], [nOutputDim])
```

Bottle allows varying dimensionality input to be forwarded through any module that accepts input of `nInputDim` dimensions, and generates output of `nOutputDim` dimensions.

Bottle can be used to forward a 4D input of varying sizes through a 2D module `b x n`. The module `Bottle(module, 2)` will accept input of shape `p x q x r x n` and outputs with the shape `p x q x r x m`. Internally Bottle will view the input of `module` as `p*q*r x n`,

and view the output as `p x q x r x m`. The numbers `p x q x r` are inferred from the input and can change for every forward/backward pass.

```
input = torch.Tensor(4, 5, 3, 10)
mlp = nn.Bottle(nn.Linear(10, 2))

> print(input:size())
  4
  5
  3
 10
[torch.LongStorage of size 4]

> print(mlp:forward(input):size())
 4
 5
 3
 2
[torch.LongStorage of size 4]
```

Table Containers

While the above containers are used for manipulating input Tensors, table containers are used for manipulating tables :
* ConcatTable
* ParallelTable

These, along with all other modules for manipulating tables can be found here.

Simple layers

Simple Modules are used for various tasks like adapting Tensor methods and providing affine transformations :

- Parameterized Modules :
  - Linear : a linear transformation ;
  - SparseLinear : a linear transformation with sparse inputs ;
  - Bilinear : a bilinear transformation with sparse inputs ;
  - PartialLinear : a linear transformation with sparse inputs with the option of only computing a subset ;
  - Add : adds a bias term to the incoming data ;
  - CAdd : a component-wise addition to the incoming data ;
  - Mul : multiply a single scalar factor to the incoming data ;
  - CMul : a component-wise multiplication to the incoming data ;
  - Euclidean : the euclidean distance of the input to `k` mean centers ;
  - WeightedEuclidean : similar to Euclidean, but additionally learns a diagonal covariance matrix ;
  - Cosine : the cosine similarity of the input to `k` mean centers ;

- Modules that adapt basic Tensor methods :
  - Copy : a copy of the input with type casting ;
  - Narrow : a narrow operation over a given dimension ;
  - Replicate : repeats input `n` times along its first dimension ;
  - Reshape : a reshape of the inputs ;
  - View : a view of the inputs ;
  - Contiguous : contiguous of the inputs ;
  - Select : a select over a given dimension ;
  - MaskedSelect : a masked select module performs the torch.maskedSelect operation ;
  - Index : a index over a given dimension ;
  - Squeeze : squeezes the input;
  - Unsqueeze : unsqueeze the input, i.e., insert singleton dimension;
  - Transpose : transposes the input ;

- Modules that adapt mathematical Tensor methods :
  - AddConstant : adding a constant ;
  - MulConstant : multiplying a constant ;
  - Max : a max operation over a given dimension ;
  - Min : a min operation over a given dimension ;
  - Mean : a mean operation over a given dimension ;

    - Sum : a sum operation over a given dimension ;
    - Exp : an element-wise exp operation ;
    - Log : an element-wise log operation ;
    - Abs : an element-wise abs operation ;
    - Power : an element-wise pow operation ;
    - Square : an element-wise square operation ;
    - Sqrt : an element-wise sqrt operation ;
    - Clamp : an element-wise clamp operation ;
    - Normalize : normalizes the input to have unit `L_p` norm ;
    - MM : matrix-matrix multiplication (also supports batches of matrices) ;

- Miscellaneous Modules :
    - BatchNormalization : mean/std normalization over the mini-batch inputs (with an optional affine transform) ;
    - PixelShuffle : Rearranges elements in a tensor of shape `[C*r, H, W]` to a tensor of shape `[C, H*r, W*r]` ;
    - Identity : forward input as-is to output (useful with ParallelTable) ;
    - Dropout : masks parts of the `input` using binary samples from a bernoulli distribution ;
    - SpatialDropout : same as Dropout but for spatial inputs where adjacent pixels are strongly correlated ;
    - VolumetricDropout : same as Dropout but for volumetric inputs where adjacent voxels are strongly correlated ;
    - Padding : adds padding to a dimension ;
    - L1Penalty : adds an L1 penalty to an input (for sparsity) ;
    - GradientReversal : reverses the gradient (to maximize an objective function) ;
    - GPU : decorates a module so that it can be executed on a specific GPU device.
    - TemporalDynamicKMaxPooling : selects the k highest values in a sequence. k can be calculated based on sequence length ;

Linear

```
module = nn.Linear(inputDimension, outputDimension, [bias = true])
```

Applies a linear transformation to the incoming data, i.e. `y = Ax + b`. The `input` tensor given in `forward(input)` must be either a vector (1D tensor) or matrix (2D tensor). If the input is a matrix, then each row is assumed to be an input sample of given batch. The layer can be used without bias by setting `bias = false`.

You can create a layer in the following way:

```
module = nn.Linear(10, 5)  -- 10 inputs, 5 outputs
```

Usually this would be added to a network of some kind, e.g.:

```
mlp = nn.Sequential()
mlp:add(module)
```

The weights and biases (*A* and *b*) can be viewed with:

```
print(module.weight)
print(module.bias)
```

The gradients for these weights can be seen with:

```
print(module.gradWeight)
print(module.gradBias)
```

As usual with `nn` modules, applying the linear transformation is performed with:

```
x = torch.Tensor(10) -- 10 inputs
y = module:forward(x)
```

SparseLinear

```
module = nn.SparseLinear(inputDimension, outputDimension)
```

Applies a linear transformation to the incoming sparse data, i.e. `y = Ax + b`. The `input` tensor given in `forward(input)` must be a sparse vector represented as 2D tensor of the form torch.Tensor(N, 2) where the pairs represent indices and values.
The SparseLinear layer is useful when the number of input dimensions is very large and the input data is sparse.

You can create a sparse linear layer in the following way:

```
module = nn.SparseLinear(10000, 2)  -- 10000 inputs, 2 outputs
```

The sparse linear module may be used as part of a larger network, and apart from the form of the input, SparseLinear operates in exactly the same way as the Linear layer.

A sparse input vector may be created as so…

```
x = torch.Tensor({ {1, 0.1}, {2, 0.3}, {10, 0.3}, {31, 0.2} })

 print(x)

  1.0000   0.1000
  2.0000   0.3000
 10.0000   0.3000
 31.0000   0.2000
[torch.Tensor of dimension 4x2]
```

The first column contains indices, the second column contains values in a a vector where all other elements are zeros. The indices should not exceed the stated dimensions of the input to the layer (10000 in the example).

Bilinear

```
module = nn.Bilinear(inputDimension1, inputDimension2,
outputDimension, [bias = true])
```

Applies a bilinear transformation to the incoming data, i.e. `\forall k: y_k = x_1 A_k x_2 + b`. The `input` tensor given in `forward(input)` is a table containing both inputs `x_1` and `x_2`, which are tensors of size `N x inputDimension1` and `N x inputDimension2`, respectively. The layer can be trained without biases by setting `bias = false`.

You can create a layer in the following way:

```
 module = nn.Bilinear(10, 5, 3)  -- 10 and 5 inputs, 3 outputs
```

Input data for this layer would look as follows:

```
 input = {torch.randn(128, 10), torch.randn(128, 5)}  -- 128 input
examples
 module:forward(input)
```

PartialLinear

```
module = nn.PartialLinear(inputSize, outputSize, [bias = true])
```

PartialLinear is a Linear layer that allows the user to a set a collection of column indices. When the column indices are set, the layer will behave like a Linear layer that only has those columns. Meanwhile, all parameters are preserved, so resetting the PartialLinear layer will result in a module that behaves just like a regular Linear layer.

This module is useful, for instance, when you want to do forward-backward on only a subset of a Linear layer during training but use the full Linear layer at test time.

You can create a layer in the following way:

```
 module = nn.PartialLinear(5, 3)  -- 5 inputs, 3 outputs
```

Input data for this layer would look as follows:

```
 input = torch.randn(128, 5)  -- 128 input examples
 module:forward(input)
```

One can set the partition of indices to compute using the function `setPartition(indices)` where `indices` is a tensor containing the indices to compute.

```
module = nn.PartialLinear(5, 3)  -- 5 inputs, 3 outputs
module:setPartition(torch.Tensor({2,4})) -- only compute the 2nd
and 4th indices out of a total of 5 indices
```

One can reset the partition via the `resetPartition()` function that resets the partition to compute all indices, making it's behaviour equivalent to `nn.Linear`

Dropout

```
module = nn.Dropout(p)
```

During training, `Dropout` masks parts of the `input` using binary samples from a bernoulli distribution.
Each `input` element has a probability of `p` of being dropped, i.e having its commensurate output element be zero. This has proven an effective technique for regularization and preventing the co-adaptation of neurons (see Hinton et al. 2012).

Furthermore, the outputs are scaled by a factor of `1/(1-p)` during training. This allows the `input` to be simply forwarded as-is during evaluation.

In this example, we demonstrate how the call to forward samples different `outputs` to dropout (the zeros) given the same `input`:

```
module = nn.Dropout()

> x = torch.Tensor{{1, 2, 3, 4}, {5, 6, 7, 8}}

> module:forward(x)
  2   0   0   8
 10   0  14   0
[torch.DoubleTensor of dimension 2x4]

> module:forward(x)
  0   0   6   0
 10   0   0   0
[torch.DoubleTensor of dimension 2x4]
```

Backward drops out the gradients at the same location:

```
> module:forward(x)
  0   4   0   0
```

```
 10  12   0  16
[torch.DoubleTensor of dimension 2x4]

> module:backward(x, x:clone():fill(1))
 0  2  0  0
 2  2  0  2
[torch.DoubleTensor of dimension 2x4]
```

In both cases the `gradOutput` and `input` are scaled by `1/(1-p)`, which in this case is `2`.

During evaluation, `Dropout` does nothing more than forward the input such that all elements of the input are considered.

```
> module:evaluate()

> module:forward(x)
 1  2  3  4
 5  6  7  8
[torch.DoubleTensor of dimension 2x4]
```

There is also an option for stochastic evaluation which drops the `outputs` just like how it is done during training:

```
module_stochastic_evaluation = nn.Dropout(nil, nil, nil, true)

> module_stochastic_evaluation:evaluate()

> module_stochastic_evaluation:forward(x)
  2   4   6   0
  0  12  14   0
[torch.DoubleTensor of dimension 2x4]
```

We can return to training our model by first calling Module:training():

```
> module:training()

> return module:forward(x)
  2   4   6   0
  0   0   0  16
[torch.DoubleTensor of dimension 2x4]
```

When used, `Dropout` should normally be applied to the input of parameterized Modules like Linear or SpatialConvolution. A `p` of `0.5` (the default) is usually okay for hidden layers. `Dropout` can sometimes be used successfully on the dataset inputs with a `p` around `0.2`. It sometimes works best following Transfer Modules like ReLU. All this depends a great deal on the dataset so its up to the user to try different combinations.

SpatialDropout

`module` = `nn.SpatialDropout(p)`

This version performs the same function as `nn.Dropout`, however it assumes the 2 right-most dimensions of the input are spatial, performs one Bernoulli trial per output feature when training, and extends this dropout value across the entire feature map.

As described in the paper "Efficient Object Localization Using Convolutional Networks" (http://arxiv.org/abs/1411.4280), if adjacent pixels within feature maps are strongly correlated (as is normally the case in early convolution layers) then iid dropout will not regularize the activations and will otherwise just result in an effective learning rate decrease. In this case, `nn.SpatialDropout` will help promote independence between feature maps and should be used instead.

`nn.SpatialDropout` accepts 3D or 4D inputs. If the input is 3D than a layout of (features x height x width) is assumed and for 4D (batch x features x height x width) is assumed.

VolumetricDropout

`module` = `nn.VolumetricDropout(p)`

This version performs the same function as `nn.Dropout`, however it assumes the 3 right-most dimensions of the input are spatial, performs one Bernoulli trial per output feature when training, and extends this dropout value across the entire feature map.

As described in the paper "Efficient Object Localization Using Convolutional Networks" (http://arxiv.org/abs/1411.4280), if adjacent voxels within feature maps are strongly correlated (as is normally the case in early convolution layers) then iid dropout will not regularize the activations and will otherwise just result in an effective learning rate decrease. In this case, `nn.VolumetricDropout` will help promote independence between feature maps and should be used instead.

`nn.VolumetricDropout` accepts 4D or 5D inputs. If the input is 4D than a layout of (features x time x height x width) is assumed and for 5D (batch x features x time x height x width) is assumed.

Abs

```
module = Abs()
```

```
m = nn.Abs()
ii = torch.linspace(-5, 5)
oo = m:forward(ii)
go = torch.ones(100)
gi = m:backward(ii, go)
gnuplot.plot({'f(x)', ii, oo, '+-'}, {'df/dx', ii, gi, '+-'})
gnuplot.grid(true)
```

Add

```
module = nn.Add(inputDimension, scalar)
```

Applies a bias term to the incoming data, i.e. `yi = x_i + b_i`, or if `scalar = true` then uses a single bias term, `yi = x_i + b`. So if `scalar = true` then `inputDimension` value will be disregarded.

Example:

```
y = torch.Tensor(5)
mlp = nn.Sequential()
mlp:add(nn.Add(5))

function gradUpdate(mlp, x, y, criterion, learningRate)
```

```
   local pred = mlp:forward(x)
   local err = criterion:forward(pred, y)
   local gradCriterion = criterion:backward(pred, y)
   mlp:zeroGradParameters()
   mlp:backward(x, gradCriterion)
   mlp:updateParameters(learningRate)
   return err
end

for i = 1, 10000 do
   x = torch.rand(5)
   y:copy(x);
   for i = 1, 5 do y[i] = y[i] + i; end
   err = gradUpdate(mlp, x, y, nn.MSECriterion(), 0.01)
end

print(mlp:get(1).bias)
```

gives the output:

```
 1.0000
 2.0000
 3.0000
 4.0000
 5.0000
[torch.Tensor of dimension 5]
```

i.e. the network successfully learns the input `x` has been shifted to produce the output `y`.

CAdd

```
module = nn.CAdd(size)
```

Applies a component-wise addition to the incoming data, i.e. `y_i = x_i + b_i`. Argument `size` can be one or many numbers (sizes) or a `torch.LongStorage`. For example, `nn.CAdd(3,4,5)` is equivalent to `nn.CAdd(torch.LongStorage{3,4,5})`. If the size for a particular dimension is 1, the addition will be expanded along the entire axis.

Example:

```
mlp = nn.Sequential()
mlp:add(nn.CAdd(5, 1))

y = torch.Tensor(5, 4)
bf = torch.Tensor(5, 4)
for i = 1, 5 do bf[i] = i; end -- scale input with this

function gradUpdate(mlp, x, y, criterion, learningRate)
   local pred = mlp:forward(x)
   local err = criterion:forward(pred, y)
   local gradCriterion = criterion:backward(pred, y)
   mlp:zeroGradParameters()
   mlp:backward(x, gradCriterion)
   mlp:updateParameters(learningRate)
   return err
end

for i = 1, 10000 do
   x = torch.rand(5, 4)
   y:copy(x)
   y:add(bf)
   err = gradUpdate(mlp, x, y, nn.MSECriterion(), 0.01)
end

print(mlp:get(1).bias)
```

gives the output:

```
 1.0000
 2.0000
 3.0000
 4.0000
 5.0000
[torch.Tensor of dimension 5x1]
```

i.e. the network successfully learns the input `x` has been shifted by those bias factors to produce the output `y`.

Mul

```
module = nn.Mul()
```

Applies a *single* scaling factor to the incoming data, i.e. `y = w x`, where `w` is a scalar.

Example:

```
y = torch.Tensor(5)
mlp = nn.Sequential()
mlp:add(nn.Mul())

function gradUpdate(mlp, x, y, criterion, learningRate)
   local pred = mlp:forward(x)
   local err = criterion:forward(pred, y)
   local gradCriterion = criterion:backward(pred, y)
   mlp:zeroGradParameters()
   mlp:backward(x, gradCriterion)
   mlp:updateParameters(learningRate)
   return err
end

for i = 1, 10000 do
   x = torch.rand(5)
   y:copy(x)
   y:mul(math.pi)
   err = gradUpdate(mlp, x, y, nn.MSECriterion(), 0.01)
end

print(mlp:get(1).weight)
```

gives the output:

```
 3.1416
[torch.Tensor of dimension 1]
```

i.e. the network successfully learns the input `x` has been scaled by pi.

CMul

```
```

```
module = nn.CMul(size)
```

Applies a component-wise multiplication to the incoming data, i.e. `y_i = w_i * x_i`.
Argument `size` can be one or many numbers (sizes) or a `torch.LongStorage`. For example, `nn.CMul(3,4,5)` is equivalent to `nn.CMul(torch.LongStorage{3,4,5})`.
If the size for a particular dimension is 1, the multiplication will be expanded along the entire axis.

Example:

```
mlp = nn.Sequential()
mlp:add(nn.CMul(5, 1))

y = torch.Tensor(5, 4)
sc = torch.Tensor(5, 4)
for i = 1, 5 do sc[i] = i; end -- scale input with this

function gradUpdate(mlp, x, y, criterion, learningRate)
   local pred = mlp:forward(x)
   local err = criterion:forward(pred, y)
   local gradCriterion = criterion:backward(pred, y)
   mlp:zeroGradParameters()
   mlp:backward(x, gradCriterion)
   mlp:updateParameters(learningRate)
   return err
end

for i = 1, 10000 do
   x = torch.rand(5, 4)
   y:copy(x)
   y:cmul(sc)
   err = gradUpdate(mlp, x, y, nn.MSECriterion(), 0.01)
end

print(mlp:get(1).weight)
```

gives the output:

```
 1.0000
 2.0000
 3.0000
 4.0000
```

```
 5.0000
[torch.Tensor of dimension 5x1]
```

i.e. the network successfully learns the input `x` has been scaled by those scaling factors to produce the output `y`.

Max

```
module = nn.Max(dimension, nInputDim)
```

Applies a max operation over dimension `dimension`.
Hence, if an `nxpxq` Tensor was given as input, and `dimension` = `2` then an `nxq` matrix would be output.
When `nInputDim` is provided, inputs larger than that value will be considered batches where the actual `dimension` to apply the max operation will be dimension `dimension + 1`.

Min

```
module = nn.Min(dimension, nInputDim)
```

Applies a min operation over dimension `dimension`.
Hence, if an `nxpxq` Tensor was given as input, and `dimension` = `2` then an `nxq` matrix would be output.
When `nInputDim` is provided, inputs larger than that value will be considered batches where the actual `dimension` to apply the min operation will be dimension `dimension + 1`.

Mean

```
module = nn.Mean(dimension, nInputDim)
```

Applies a mean operation over dimension `dimension`.
Hence, if an `nxpxq` Tensor was given as input, and `dimension` = `2` then an `nxq` matrix would be output.
When `nInputDim` is provided, inputs larger than that value will be considered batches where the actual `dimension` to apply the sum operation will be dimension `dimension + 1`.
This module is based on nn.Sum.

Sum

```
module = nn.Sum(dimension, nInputDim, sizeAverage)
```

Applies a sum operation over dimension `dimension`.
Hence, if an `nxpxq` Tensor was given as input, and `dimension` = `2` then an `nxq` matrix would be output.
When `nInputDim` is provided, inputs larger than that value will be considered batches where the actual `dimension` to apply the sum operation will be dimension `dimension + 1`.
Negative indexing is allowed by providing a negative value to `nInputDim`.
When `sizeAverage` is provided, the sum is divided by the size of the input in this `dimension`. This is equivalent to the mean operation performed by the nn.Mean module.

Euclidean

```
module = nn.Euclidean(inputSize,outputSize)
```

Outputs the Euclidean distance of the input to `outputSize` centers, i.e. this layer has the weights `w_j`, for `j` = `1`,.., `outputSize`, where `w_j` are vectors of dimension `inputSize`.

The distance `y_j` between center `j` and input `x` is formulated as `y_j = || w_j - x ||`.

WeightedEuclidean

```
module = nn.WeightedEuclidean(inputSize,outputSize)
```

This module is similar to Euclidean, but additionally learns a separate diagonal covariance matrix across the features of the input space *for each center*.

In other words, for each of the `outputSize` centers `w_j`, there is a diagonal covariance matrices `c_j`, for `j` = `1`,.., `outputSize`, where `c_j` are stored as vectors of size `inputSize`.

The distance `y_j` between center `j` and input `x` is formulated as `y_j = || c_j * (w_j - x) ||`.

Cosine

```
module = nn.Cosine(inputSize,outputSize)
```

Outputs the cosine similarity of the input to `outputSize` centers, i.e. this layer has the weights `w_j`, for `j` = `1`,.., `outputSize`, where `w_j` are vectors of dimension `inputSize`.

The distance `y_j` between center `j` and input `x` is formulated as `y_j = (x · w_j) / ( || w_j || * || x || )`.

Identity

```
module = nn.Identity()
```

Creates a module that returns whatever is input to it as output.
This is useful when combined with the module ParallelTable in case you do not wish to do anything to one of the input Tensors.

Example:

```
mlp = nn.Identity()
print(mlp:forward(torch.ones(5, 2)))
```

gives the output:

```
 1  1
 1  1
 1  1
 1  1
 1  1
[torch.Tensor of dimension 5x2]
```

Here is a more useful example, where one can implement a network which also computes a `Criterion` using this module:

```
pred_mlp = nn.Sequential()  -- A network that makes predictions
given x.
pred_mlp:add(nn.Linear(5, 4))
pred_mlp:add(nn.Linear(4, 3))

xy_mlp = nn.ParallelTable() -- A network for predictions and for
keeping the
xy_mlp:add(pred_mlp)        -- true label for comparison with a
criterion
xy_mlp:add(nn.Identity())   -- by forwarding both x and y through
the network.

mlp = nn.Sequential()       -- The main network that takes both x
and y.
mlp:add(xy_mlp)             -- It feeds x and y to parallel
networks;
cr = nn.MSECriterion()
cr_wrap = nn.CriterionTable(cr)
mlp:add(cr_wrap)            -- and then applies the criterion.

for i = 1, 100 do           -- Do a few training iterations
   x = torch.ones(5)        -- Make input features.
   y = torch.Tensor(3)
   y:copy(x:narrow(1,1,3))  -- Make output label.
   err = mlp:forward{x,y}   -- Forward both input and output.
   print(err)               -- Print error from criterion.

   mlp:zeroGradParameters() -- Do backprop...
   mlp:backward({x, y})
   mlp:updateParameters(0.05)
```

```
end
```

Copy

```
module = nn.Copy(inputType, outputType, [forceCopy, dontCast])
```

This layer copies the input to output with type casting from `inputType` to `outputType`. Unless `forceCopy` is true, when the first two arguments are the same, the input isn't copied, only transferred as the output.
The default `forceCopy` is false.
When `dontCast` is true, a call to `nn.Copy:type(type)` will not cast the module's `output` and `gradInput` `Tensor`s to the new type.
The default is false.

Narrow

```
module = nn.Narrow(dimension, offset, length)
```

Narrow is application of narrow operation in a module. The module further supports negative `length`, `dim` and `offset` to handle inputs of unknown size.

```
> x = torch.rand(4, 5)

> x
 0.3695  0.2017  0.4485  0.4638  0.0513
 0.9222  0.1877  0.3388  0.6265  0.5659
 0.8785  0.7394  0.8265  0.9212  0.0129
 0.2290  0.7971  0.2113  0.1097  0.3166
[torch.DoubleTensor of size 4x5]

> nn.Narrow(1, 2, 3):forward(x)
 0.9222  0.1877  0.3388  0.6265  0.5659
 0.8785  0.7394  0.8265  0.9212  0.0129
 0.2290  0.7971  0.2113  0.1097  0.3166
```

```
[torch.DoubleTensor of size 3x5]

> nn.Narrow(1, 2, -1):forward(x)
 0.9222  0.1877  0.3388  0.6265  0.5659
 0.8785  0.7394  0.8265  0.9212  0.0129
 0.2290  0.7971  0.2113  0.1097  0.3166
[torch.DoubleTensor of size 3x5]

> nn.Narrow(1, 2, 2):forward(x)
 0.9222  0.1877  0.3388  0.6265  0.5659
 0.8785  0.7394  0.8265  0.9212  0.0129
[torch.DoubleTensor of size 2x5]

> nn.Narrow(1, 2, -2):forward(x)
 0.9222  0.1877  0.3388  0.6265  0.5659
 0.8785  0.7394  0.8265  0.9212  0.0129
[torch.DoubleTensor of size 2x5]

> nn.Narrow(2, 2, 3):forward(x)
 0.2017  0.4485  0.4638
 0.1877  0.3388  0.6265
 0.7394  0.8265  0.9212
 0.7971  0.2113  0.1097
[torch.DoubleTensor of size 4x3]

> nn.Narrow(2, 2, -2):forward(x)
 0.2017  0.4485  0.4638
 0.1877  0.3388  0.6265
 0.7394  0.8265  0.9212
 0.7971  0.2113  0.1097
[torch.DoubleTensor of size 4x3]
```

Replicate

```
module = nn.Replicate(nFeature [, dim, ndim])
```

This class creates an output where the input is replicated `nFeature` times along dimension `dim` (default 1).
There is no memory allocation or memory copy in this module.

It sets the stride along the `dim` th dimension to zero.
When provided, `ndim` should specify the number of non-batch dimensions.
This allows the module to replicate the same non-batch dimension `dim` for both batch and non-batch `inputs` .

```
> x = torch.linspace(1, 5, 5)
 1
 2
 3
 4
 5
[torch.DoubleTensor of dimension 5]

> m = nn.Replicate(3)
> o = m:forward(x)
 1  2  3  4  5
 1  2  3  4  5
 1  2  3  4  5
[torch.DoubleTensor of dimension 3x5]

> x:fill(13)
 13
 13
 13
 13
 13
[torch.DoubleTensor of dimension 5]

> print(o)
 13  13  13  13  13
 13  13  13  13  13
 13  13  13  13  13
[torch.DoubleTensor of dimension 3x5]
```

Reshape

```
module = nn.Reshape(dimension1, dimension2, ... [, batchMode])
```

Reshapes an `nxpxqx..` `Tensor` into a `dimension1xdimension2x...` `Tensor` , taking

the elements row-wise.

The optional last argument `batchMode`, when `true` forces the first dimension of the input to be considered the batch dimension, and thus keep its size fixed.
This is necessary when dealing with batch sizes of one.
When `false`, it forces the entire input (including the first dimension) to be reshaped to the input size.
Default `batchMode=nil`, which means that the module considers inputs with more elements than the produce of provided sizes, i.e. `dimension1xdimension2x...`, to be batches.

Example:

```
> x = torch.Tensor(4,4)
> for i = 1, 4 do
>    for j = 1, 4 do
>       x[i][j] = (i-1)*4+j
>    end
> end
> print(x)

  1   2   3   4
  5   6   7   8
  9  10  11  12
 13  14  15  16
[torch.Tensor of dimension 4x4]

> print(nn.Reshape(2,8):forward(x))

  1   2   3   4   5   6   7   8
  9  10  11  12  13  14  15  16
[torch.Tensor of dimension 2x8]

> print(nn.Reshape(8,2):forward(x))

  1   2
  3   4
  5   6
  7   8
  9  10
 11  12
 13  14
 15  16
[torch.Tensor of dimension 8x2]

> print(nn.Reshape(16):forward(x))
```

```
  1
  2
  3
  4
  5
  6
  7
  8
  9
 10
 11
 12
 13
 14
 15
 16
[torch.Tensor of dimension 16]

> y = torch.Tensor(1, 4):fill(0)
> print(y)

 0  0  0  0
 [torch.DoubleTensor of dimension 1x4]

> print(nn.Reshape(4):forward(y))

 0  0  0  0
 [torch.DoubleTensor of dimension 1x4]

> print(nn.Reshape(4, false):forward(y))

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

View

```
module = nn.View(sizes)
```

This module creates a new view of the input tensor using the `sizes` passed to the constructor.
The parameter `sizes` can either be a `LongStorage` or numbers.
The method `setNumInputDims()` allows to specify the expected number of dimensions of the inputs of the modules.
This makes it possible to use minibatch inputs when using a size `-1` for one of the dimensions.
The method `resetSize(sizes)` allows to reset the view size of the module after initialization.

Example 1:

```
> x = torch.Tensor(4, 4)
> for i = 1, 4 do
>    for j = 1, 4 do
>       x[i][j] = (i-1)*4+j
>    end
> end
> print(x)

  1   2   3   4
  5   6   7   8
  9  10  11  12
 13  14  15  16
[torch.Tensor of dimension 4x4]

> print(nn.View(2, 8):forward(x))

  1   2   3   4   5   6   7   8
  9  10  11  12  13  14  15  16
[torch.DoubleTensor of dimension 2x8]

> print(nn.View(torch.LongStorage{8,2}):forward(x))

  1   2
  3   4
  5   6
  7   8
  9  10
 11  12
 13  14
 15  16
[torch.DoubleTensor of dimension 8x2]
```

```
> print(nn.View(16):forward(x))

  1
  2
  3
  4
  5
  6
  7
  8
  9
 10
 11
 12
 13
 14
 15
 16
[torch.DoubleTensor of dimension 16]
```

Example 2:

```
> input = torch.Tensor(2, 3)
> minibatch = torch.Tensor(5, 2, 3)
> m = nn.View(-1):setNumInputDims(2)
> print(#m:forward(input))

 6
[torch.LongStorage of size 1]

> print(#m:forward(minibatch))

 5
 6
[torch.LongStorage of size 2]
```

Contiguous

```
module = nn.Contiguous()
```

Is used to make `input`, `gradOutput` or both contiguous, corresponds to `torch.contiguous` function.
Only does copy and allocation if `input` or `gradOutput` is not contiguous, otherwise passes the same `Tensor`.

Select

```
module = nn.Select(dim, index)
```

Selects a dimension and index of a `nxpxqx..` `Tensor`.

Example:

```
mlp = nn.Sequential()
mlp:add(nn.Select(1, 3))

x = torch.randn(10, 5)
print(x)
print(mlp:forward(x))
```

gives the output:

```
 0.9720 -0.0836  0.0831 -0.2059 -0.0871
 0.8750 -2.0432 -0.1295 -2.3932  0.8168
 0.0369  1.1633  0.6483  1.2862  0.6596
 0.1667 -0.5704 -0.7303  0.3697 -2.2941
 0.4794  2.0636  0.3502  0.3560 -0.5500
-0.1898 -1.1547  0.1145 -1.1399  0.1711
-1.5130  1.4445  0.2356 -0.5393 -0.6222
-0.6587  0.4314  1.1916 -1.4509  1.9400
 0.2733  1.0911  0.7667  0.4002  0.1646
 0.5804 -0.5333  1.1621  1.5683 -0.1978
[torch.Tensor of dimension 10x5]

 0.0369
```

```
 1.1633
 0.6483
 1.2862
 0.6596
[torch.Tensor of dimension 5]
```

This can be used in conjunction with Concat to emulate the behavior of Parallel, or to select various parts of an input Tensor to perform operations on. Here is a fairly complicated example:

```
mlp = nn.Sequential()
c = nn.Concat(2)
for i = 1, 10 do
   local t = nn.Sequential()
   t:add(nn.Select(1, i))
   t:add(nn.Linear(3, 2))
   t:add(nn.Reshape(2, 1))
   c:add(t)
end
mlp:add(c)

pred = mlp:forward(torch.randn(10, 3))
print(pred)

for i = 1, 10000 do     -- Train for a few iterations
   x = torch.randn(10, 3)
   y = torch.ones(2, 10)
   pred = mlp:forward(x)

   criterion = nn.MSECriterion()
   err = criterion:forward(pred, y)
   gradCriterion = criterion:backward(pred, y)
   mlp:zeroGradParameters()
   mlp:backward(x, gradCriterion)
   mlp:updateParameters(0.01)
   print(err)
end
```

MaskedSelect

```
module = nn.MaskedSelect()
```

Performs a torch.MaskedSelect on a `Tensor`.
The mask is supplied as a tabular argument with the input on the forward and backward passes.

Example:

```
ms = nn.MaskedSelect()
mask = torch.ByteTensor({{1, 0}, {0, 1}})
input = torch.DoubleTensor({{10, 20}, {30, 40}})
print(input)
print(mask)
out = ms:forward({input, mask})
print(out)
gradIn = ms:backward({input, mask}, out)
print(gradIn[1])
```

Gives the output:

```
10  20
30  40
[torch.DoubleTensor of size 2x2]

1  0
0  1
[torch.ByteTensor of size 2x2]

10
40
[torch.DoubleTensor of size 2]

10  0
0  40
[torch.DoubleTensor of size 2x2]
```

Index

```
module = nn.Index(dim)
```

Applies the `Tensor` index operation along the given dimension. So

```
nn.Index(dim):forward{t,i}
```

gives the same output as

```
t:index(dim, i)
```

Squeeze

```
module = nn.Squeeze([dim, numInputDims])
```

Applies the `Tensor` squeeze operation. So

```
nn.Squeeze():forward(t)
```

gives the same output as

```
t:squeeze()
```

Setting `numInputDims` allows to use this module on batches.

Unsqueeze

```
module = nn.Unsqueeze(pos [, numInputDims])
```

Insert singleton dim (i.e., dimension 1) at position `pos`.

For an `input` with `dim = input:dim()`, there are `dim + 1` possible positions to insert the singleton dimension.
For example, if `input` is `3` dimensional `Tensor` in size `p x q x r`, then the singleton dim can be inserted at the following `4` positions

```
pos = 1: 1 x p x q x r
pos = 2: p x 1 x q x r
pos = 3: p x q x 1 x r
pos = 4: p x q x r x 1
```

Example:

```
input = torch.Tensor(2, 4, 3) -- input: 2 x 4 x 3

-- insert at head
m = nn.Unsqueeze(1)
m:forward(input) -- output: 1 x 2 x 4 x 3

-- insert at tail
m = nn.Unsqueeze(4)
m:forward(input) -- output: 2 x 4 x 3 x 1

-- insert in between
m = nn.Unsqueeze(2)
m:forward(input) -- output: 2 x 1 x 4 x 3

-- the input size can vary across calls
input2 = torch.Tensor(3, 5, 7) -- input2: 3 x 5 x 7
m:forward(input2) -- output: 3 x 1 x 5 x 7
```

Indicate the expected input feature map dimension by specifying `numInputDims`.
This allows the module to work with mini-batch. Example:

```
b = 5 -- batch size 5
input = torch.Tensor(b, 2, 4, 3) -- input: b x 2 x 4 x 3
numInputDims = 3 -- input feature map should be the last 3 dims

m = nn.Unsqueeze(4, numInputDims)
m:forward(input) -- output: b x 2 x 4 x 3 x 1

m = nn.Unsqueeze(2):setNumInputDims(numInputDims)
m:forward(input) -- output: b x 2 x 1 x 4 x 3
```

Transpose

```
module = nn.Transpose({dim1, dim2} [, {dim3, dim4}, ...])
```

Swaps dimension `dim1` with `dim2`, then `dim3` with `dim4`, and so on. So

```
nn.Transpose({dim1, dim2}, {dim3, dim4}):forward(t)
```

gives the same output as

```
t:transpose(dim1, dim2)
t:transpose(dim3, dim4)
```

Exp

```
module = nn.Exp()
```

Applies the `exp` function element-wise to the input `Tensor`, thus outputting a `Tensor` of the same dimension.

```
ii = torch.linspace(-2, 2)
m = nn.Exp()
oo = m:forward(ii)
go = torch.ones(100)
gi = m:backward(ii,go)
gnuplot.plot({'f(x)', ii, oo, '+-'}, {'df/dx', ii, gi, '+-'})
gnuplot.grid(true)
```

Log

```
module = nn.Log()
```

Applies the `log` function element-wise to the input `Tensor`, thus outputting a Tensor of the same dimension.

Square

```
module = nn.Square()
```

Takes the square of each element.

```
ii = torch.linspace(-5, 5)
m = nn.Square()
oo = m:forward(ii)
go = torch.ones(100)
gi = m:backward(ii, go)
gnuplot.plot({'f(x)', ii, oo, '+-'}, {'df/dx', ii, gi, '+-'})
gnuplot.grid(true)
```

Sqrt

```
module = nn.Sqrt()
```

Takes the square root of each element.

```
ii = torch.linspace(0, 5)
```

```
m = nn.Sqrt()
oo = m:forward(ii)
go = torch.ones(100)
gi = m:backward(ii, go)
gnuplot.plot({'f(x)', ii, oo, '+-'}, {'df/dx', ii, gi, '+-'})
gnuplot.grid(true)
```

Power

```
module = nn.Power(p)
```

Raises each element to its `p` -th power.

```
ii = torch.linspace(0, 2)
m = nn.Power(1.25)
oo = m:forward(ii)
go = torch.ones(100)
gi = m:backward(ii, go)
gnuplot.plot({'f(x)', ii, oo, '+-'}, {'df/dx', ii, gi, '+-'})
gnuplot.grid(true)
```

Clamp

```
module = nn.Clamp(min_value, max_value)
```

Clamps all elements into the range `[min_value, max_value]` .
Output is identical to input in the range, otherwise elements less than `min_value` (or greater than `max_value`) are saturated to `min_value` (or `max_value`).

```
A = torch.randn(2, 5)
m = nn.Clamp(-0.1, 0.5)
B = m:forward(A)

print(A)  -- input
-1.1321  0.0227 -0.4672  0.6519 -0.5380
 0.9061 -1.0858  0.3697 -0.8120 -1.6759
[torch.DoubleTensor of size 3x5]

print(B)  -- output
-0.1000  0.0227 -0.1000  0.5000 -0.1000
 0.5000 -0.1000  0.3697 -0.1000 -0.1000
[torch.DoubleTensor of size 3x5]
```

Normalize

```
module = nn.Normalize(p, [eps])
```

Normalizes the input `Tensor` to have unit `L_p` norm. The smoothing parameter `eps` prevents division by zero when the input contains all zero elements (default = `1e-10`).

Input can be 1D or 2D (in which case it's considered as in batch mode)

```
A = torch.randn(3, 5)
m = nn.Normalize(2)
B = m:forward(A) -- B is also 3 x 5
-- take the L2 norm over the second axis:
print(torch.norm(B, 2, 2)) -- norms is [1, 1, 1]
```

`Normalize` has a specialized implementation for the `inf` norm, which corresponds to the maximum norm.

```
A = torch.randn(3,5)
m = nn.Normalize(math.huge) -- uses maximum/inf norm
B = m:forward(A)
maxA = torch.abs(A):max(2)
print(A,B,maxA)
```

MM

```
module = nn.MM(transA, transB)
```

Performs multiplications on one or more pairs of matrices. If `transA` is set to true, the first matrix is transposed before multiplication. If `transB` is set to true, the second matrix is transposed before multiplication. By default, the matrices do not get transposed.

The module also accepts 3D inputs which are interpreted as batches of matrices. When using batches, the first input matrix should be of size `b x m x n` and the second input matrix should be of size `b x n x p` (assuming `transA` and `transB` are not set). If `transA` or `transB` is set, transpose takes place between the second and the third dimensions for the corresponding matrix.

```
model = nn.MM()
A = torch.randn(b, m, n)
B = torch.randn(b, n, p)
C = model:forward({A, B})  -- C will be of size `b x m x p`

model = nn.MM(true, false)
A = torch.randn(b, n, m)
B = torch.randn(b, n, p)
C = model:forward({A, B})  -- C will be of size `b x m x p`
```

BatchNormalization

```
module = nn.BatchNormalization(N [, eps] [, momentum] [,affine])
```

where `N` is the dimensionality of input
`eps` is a small value added to the standard-deviation to avoid divide-by-zero. Defaults to `1e-5`.
`affine` is a boolean. When set to false, the learnable affine transform is disabled. Defaults to

true

During training, this layer keeps a running estimate of its computed mean and std.
The running sum is kept with a default momentum of 0.1 (unless over-ridden)
During evaluation, this running mean/std is used for normalization.

Implements Batch Normalization as described in the paper: “Batch Normalization: Accelerating Deep Network Training by Reducing Internal Covariate Shift” by Sergey Ioffe, Christian Szegedy.

The operation implemented is:

```
                x - mean(x)
y =  ----------------------------- * gamma + beta
      standard-deviation(x) + eps
```

where the mean and standard-deviation are calculated per-dimension over the mini-batches and where gamma and beta are learnable parameter vectors of size `N` (where `N` is the input size).
The learning of gamma and beta is optional.
The module only accepts 2D inputs.

```
-- with learnable parameters
model = nn.BatchNormalization(m)
A = torch.randn(b, m)
C = model:forward(A)  -- C will be of size `b x m`

-- without learnable parameters
model = nn.BatchNormalization(m, nil, nil, false)
A = torch.randn(b, m)
C = model:forward(A)  -- C will be of size `b x m`
```

PixelShuffle

`module = nn.PixelShuffle(r)`

Rearranges elements in a tensor of shape `[C*r, H, W]` to a tensor of shape `[C, H*r, W*r]`. This is useful for implementing efficient sub-pixel convolution with a stride of `1/r` (see Shi et. al). Below we show how the `PixelShuffle` module can be used to learn upscaling

filters to transform a low-resolution input to a high resolution one, with a 3x upscale factor. This is useful for tasks such as super-resolution, see "Real-Time Single Image and Video Super-Resolution Using an Efficient Sub-Pixel Convolutional Neural Network" - Shi et al. for further details.

```
upscaleFactor = 3
inputChannels = 1

model = nn.Sequential()
model:add(nn.SpatialConvolution(inputChannels, 64, 5, 5, 1, 1, 2,
2))
model:add(nn.ReLU())

model:add(nn.SpatialConvolution(64, 32, 3, 3, 1, 1, 1, 1))
model:add(nn.ReLU())

model:add(nn.SpatialConvolution(32, inputChannels * upscaleFactor *
upscaleFactor, 3, 3, 1, 1, 1, 1))
model:add(nn.PixelShuffle(upscaleFactor))

input = torch.Tensor(1, 192, 256);
out = model:forward(input)
out:size()
   1
 576
 768
[torch.LongStorage of size 3]
```

Padding

```
module = nn.Padding(dim, pad [, nInputDim, value, index])
```

This module adds `pad` units of padding to dimension `dim` of the input.
If `pad` is negative, padding is added to the left, otherwise, it is added to the right of the dimension. When `nInputDim` is provided, inputs larger than that value will be considered batches where the actual `dim` to be padded will
be dimension `dim + 1`. When `value` is provide, the padding will be filled with that `value`.
The default `value` is zero.

When `index` is provided, padding will be added at that offset from the left or right, depending on the sign of `pad`.

Example 1:

```
module = nn.Padding(1, 2, 1, -1) --pad right x2
module:forward(torch.randn(3)) --non-batch input
 0.2008
 0.4848
-1.0783
-1.0000
-1.0000
[torch.DoubleTensor of dimension 5]
```

Example 2:

```
module = nn.Padding(1, -2, 1, -1) --pad left x2
module:forward(torch.randn(2, 3)) --batch input
-1.0000 -1.0000  1.0203  0.2704 -1.6164
-1.0000 -1.0000 -0.2219 -0.6529 -1.9218
[torch.DoubleTensor of dimension 2x5]
```

Example 3:

```
module = nn.Padding(1, -2, 1, -1, 2) --pad left x2, offset to index
2
module:forward(torch.randn(2, 3)) --batch input
 1.0203 -1.0000 -1.0000  0.2704 -1.6164
-0.6529 -1.0000 -1.0000 -0.2219 -1.9218
[torch.DoubleTensor of dimension 2x5]
```

L1Penalty

```
penalty = nn.L1Penalty(L1weight, sizeAverage)
```

L1Penalty is an inline module that in its forward propagation copies the input Tensor directly to the output, and computes an L1 loss of the latent state (input) and stores it in the module's

`loss` field.
During backward propagation: `gradInput = gradOutput + gradLoss`.

This module can be used in autoencoder architectures to apply L1 losses to internal latent state without having to use Identity and parallel containers to carry the internal code to an output criterion.

Example (sparse autoencoder, note: decoder should be normalized):

```
encoder = nn.Sequential()
encoder:add(nn.Linear(3, 128))
encoder:add(nn.Threshold())
decoder = nn.Linear(128, 3)

autoencoder = nn.Sequential()
autoencoder:add(encoder)
autoencoder:add(nn.L1Penalty(l1weight))
autoencoder:add(decoder)

criterion = nn.MSECriterion()  -- To measure reconstruction error
-- ...
```

GradientReversal

```
module = nn.GradientReversal([lambda = 1])
```

This module preserves the input, but takes the gradient from the subsequent layer, multiplies it by `-lambda` and passes it to the preceding layer. This can be used to maximise an objective function whilst using gradient descent, as described in “Domain-Adversarial Training of Neural Networks” (http://arxiv.org/abs/1505.07818).

One can also call:

```
module:setLambda(lambda)
```

to set the hyper-parameter `lambda` dynamically during training.

GPU

```
gpu = nn.GPU(module, device, [outdevice])
require 'cunn'
gpu:cuda()
```

Decorates an encapsulated `module` so that it can be executed on a specific GPU `device`.
The decorated module's `parameters` are thus hosted on the specified GPU `device`.
All operations on the `gpu` module are executed on that device.
Calls to `forward` / `backward` will transfer arguments `input` and `gradOutput` to the specified `device`,
which are then fed as arguments to the decorated `module`.
Returned `output` is located on the specified `outdevice` (defaults to `device`).
Returned `gradInput` is allocated on the same device as the `input`.

When serialized/deserialized, the `gpu` module will be run on the same `device` that it was serialized with.
To prevent this from happening, the module can be converted to float/double before serialization:

```
gpu:float()
gpustr = torch.serialize(gpu)
```

The module is located in the **nn** package instead of **cunn** as this allows
it to be used in CPU-only environments, which are common for production models.

The module supports nested table `input` and `gradOutput` tensors originating from multiple devices.
Each nested tensor in the returned `gradInput` will be transferred to the device its commensurate tensor in the `input`.

The intended use-case is not for model-parallelism where the models are executed in parallel on multiple devices, but
for sequential models where a single GPU doesn't have enough memory.

Example using 4 GPUs:

```
mlp = nn.Sequential()
   :add(nn.GPU(nn.Linear(10000,10000), 1))
   :add(nn.GPU(nn.Linear(10000,10000), 2))
```

```
   :add(nn.GPU(nn.Linear(10000,10000), 3))
   :add(nn.GPU(nn.Linear(10000,10000), 4, cutorch.getDevice()))
```

Note how the last `GPU` instance will return an `output` tensor on the same device as the current device (`cutorch.getDevice`).

TemporalDynamicKMaxPooling

```
module = nn.TemporalDynamicKMaxPooling(minK, [factor])
```

Selects the highest `k` values for each feature in the feature map sequence provided. The input sequence is composed of `nInputFrame` frames (i.e. `nInputFrame` is sequence length). The `input` tensor in `forward(input)` is expected to be a 2D tensor (`nInputFrame x inputFrameSize`) or a 3D tensor (`nBatchFrame x nInputFrame x inputFrameSize`), where `inputFrameSize` is the number of features across the sequence.

If `factor` is not provided, `k = minK` , else the value of k is calculated with:

```
k = math.max(minK, math.ceil(factor*nInputFrame)))
```

Table Layers

This set of modules allows the manipulation of `table` s through the layers of a neural network. This allows one to build very rich architectures:

- `table` Container Modules encapsulate sub-Modules:
  - `ConcatTable` : applies each member module to the same input `Tensor` and outputs a `table` ;
  - `ParallelTable` : applies the `i` -th member module to the `i` -th input and outputs a `table` ;
  - `MapTable` : applies a single module to every input and outputs a `table` ;
- Table Conversion Modules convert between `table` s and `Tensor` s or `table` s:
  - `SplitTable` : splits a `Tensor` into a `table` of `Tensor` s;
  - `JoinTable` : joins a `table` of `Tensor` s into a `Tensor` ;
  - `MixtureTable` : mixture of experts weighted by a gater;
  - `SelectTable` : select one element from a `table` ;
  - `NarrowTable` : select a slice of elements from a `table` ;
  - `FlattenTable` : flattens a nested `table` hierarchy;
- Pair Modules compute a measure like distance or similarity from a pair (`table`) of input `Tensor` s:
  - `PairwiseDistance` : outputs the `p` -norm. distance between inputs;
  - `DotProduct` : outputs the dot product (similarity) between inputs;
  - `CosineDistance` : outputs the cosine distance between inputs;
- CMath Modules perform element-wise operations on a `table` of `Tensor` s:
  - `CAddTable` : addition of input `Tensor` s;
  - `CSubTable` : substraction of input `Tensor` s;
  - `CMulTable` : multiplication of input `Tensor` s;
  - `CDivTable` : division of input `Tensor` s;
  - `CMaxTable` : max of input `Tensor` s;
  - `CMinTable` : min of input `Tensor` s;
- `Table` of Criteria:
  - `CriterionTable` : wraps a Criterion so that it can accept a `table` of inputs.

`table` -based modules work by supporting `forward()` and `backward()` methods that can accept `table` s as inputs.
It turns out that the usual `Sequential` module can do this, so all that is needed is other child modules that take advantage of such `table` s.

```
mlp = nn.Sequential()
t = {x, y, z}
pred = mlp:forward(t)
pred = mlp:forward{x, y, z}      -- This is equivalent to the line
before
```

ConcatTable

```
module = nn.ConcatTable()
```

`ConcatTable` is a container module that applies each member module to the same input `Tensor` or `table`.

```
                 +-----------+
             +---> {member1, |
+-------+    |   |           |
| input +----+---->  member2, |
+-------+    |   |           |
   or        +---->  member3} |
 {input}           +-----------+
```

Example 1

```
mlp = nn.ConcatTable()
mlp:add(nn.Linear(5, 2))
mlp:add(nn.Linear(5, 3))

pred = mlp:forward(torch.randn(5))
for i, k in ipairs(pred) do print(i, k) end
```

which gives the output:

```
1
```

```
-0.4073
 0.0110
[torch.Tensor of dimension 2]

2
 0.0027
-0.0598
-0.1189
[torch.Tensor of dimension 3]
```

Example 2

```
mlp = nn.ConcatTable()
mlp:add(nn.Identity())
mlp:add(nn.Identity())

pred = mlp:forward{torch.randn(2), {torch.randn(3)}}
print(pred)
```

which gives the output (using th):

```
{
  1 :
    {
      1 : DoubleTensor - size: 2
      2 :
        {
          1 : DoubleTensor - size: 3
        }
    }
  2 :
    {
      1 : DoubleTensor - size: 2
      2 :
        {
          1 : DoubleTensor - size: 3
        }
    }
}
```

ParallelTable

```
module = nn.ParallelTable()
```

`ParallelTable` is a container module that, in its `forward()` method, applies the `i`-th member module to the `i`-th input, and outputs a `table` of the set of outputs.

```
+----------+         +-----------+
| {input1, +---------> {member1, |
|          |         |           |
|  input2, +--------->  member2, |
|          |         |           |
|  input3} +--------->  member3} |
+----------+         +-----------+
```

Example

```
mlp = nn.ParallelTable()
mlp:add(nn.Linear(10, 2))
mlp:add(nn.Linear(5, 3))

x = torch.randn(10)
y = torch.rand(5)

pred = mlp:forward{x, y}
for i, k in pairs(pred) do print(i, k) end
```

which gives the output:

```
1
 0.0331
 0.7003
[torch.Tensor of dimension 2]

2
 0.0677
```

```
-0.1657
-0.7383
[torch.Tensor of dimension 3]
```

MapTable

```
module = nn.MapTable(m, share)
```

`MapTable` is a container for a single module which will be applied to all input elements. The member module is cloned as necessary to process all input elements. Call `resize(n)` to set the number of clones manually or call `clearState()` to discard all clones.

Optionally, the module can be initialized with the contained module and with a list of parameters that are shared across all clones. By default, these parameters are `weight`, `bias`, `gradWeight` and `gradBias`.

```
+----------+         +-----------+
| {input1, +---------> {member,  |
|          |         |           |
|  input2, +--------->  clone,   |
|          |         |           |
|  input3} +--------->  clone}   |
+----------+         +-----------+
```

Example

```
map = nn.MapTable()
map:add(nn.Linear(10, 3))

x1 = torch.rand(10)
x2 = torch.rand(10)
y = map:forward{x1, x2}

for i, k in pairs(y) do print(i, k) end
```

which gives the output:

```
1
 0.0345
 0.8695
 0.6502
[torch.DoubleTensor of size 3]

2
 0.0269
 0.4953
 0.2691
[torch.DoubleTensor of size 3]
```

SplitTable

```
module = SplitTable(dimension, nInputDims)
```

Creates a module that takes a `Tensor` as input and outputs several `table`s, splitting the `Tensor` along the specified `dimension`.
In the diagram below, `dimension` is equal to `1`.

```
    +----------+         +-----------+
    | input[1] +---------> {member1, |
  +----------+-+         |           |
  | input[2] +----------->  member2, |
+----------+-+           |           |
| input[3] +------------->  member3} |
+----------+             +-----------+
```

The optional parameter `nInputDims` allows to specify the number of dimensions that this module will receive.
This makes it possible to forward both minibatch and non-minibatch `Tensor`s through the same module.

Example 1

```
mlp = nn.SplitTable(2)
x = torch.randn(4, 3)
pred = mlp:forward(x)
for i, k in ipairs(pred) do print(i, k) end
```

gives the output:

```
1
 1.3885
 1.3295
 0.4281
-1.0171
[torch.Tensor of dimension 4]

2
-1.1565
-0.8556
-1.0717
-0.8316
[torch.Tensor of dimension 4]

3
-1.3678
-0.1709
-0.0191
-2.5871
[torch.Tensor of dimension 4]
```

Example 2

```
mlp = nn.SplitTable(1)
pred = mlp:forward(torch.randn(4, 3))
for i, k in ipairs(pred) do print(i, k) end
```

gives the output:

```
1
 1.6114
```

```
 0.9038
 0.8419
[torch.Tensor of dimension 3]

2
 2.4742
 0.2208
 1.6043
[torch.Tensor of dimension 3]

3
 1.3415
 0.2984
 0.2260
[torch.Tensor of dimension 3]

4
 2.0889
 1.2309
 0.0983
[torch.Tensor of dimension 3]
```

Example 3

```
mlp = nn.SplitTable(1, 2)
pred = mlp:forward(torch.randn(2, 4, 3))
for i, k in ipairs(pred) do print(i, k) end
pred = mlp:forward(torch.randn(4, 3))
for i, k in ipairs(pred) do print(i, k) end
```

gives the output:

```
1
-1.3533  0.7448 -0.8818
-0.4521 -1.2463  0.0316
[torch.DoubleTensor of dimension 2x3]

2
 0.1130 -1.3904  1.4620
 0.6722  2.0910 -0.2466
```

```
[torch.DoubleTensor of dimension 2x3]

3
 0.4672 -1.2738  1.1559
 0.4664  0.0768  0.6243
[torch.DoubleTensor of dimension 2x3]

4
 0.4194  1.2991  0.2241
 2.9786 -0.6715  0.0393
[torch.DoubleTensor of dimension 2x3]


1
-1.8932
 0.0516
-0.6316
[torch.DoubleTensor of dimension 3]

2
-0.3397
-1.8881
-0.0977
[torch.DoubleTensor of dimension 3]

3
 0.0135
 1.2089
 0.5785
[torch.DoubleTensor of dimension 3]

4
-0.1758
-0.0776
-1.1013
[torch.DoubleTensor of dimension 3]
```

The module also supports indexing from the end using negative dimensions. This allows to use this module when the number of dimensions of the input is unknown.

Example

```
```

```
m = nn.SplitTable(-2)
out = m:forward(torch.randn(3, 2))
for i, k in ipairs(out) do print(i, k) end
out = m:forward(torch.randn(1, 3, 2))
for i, k in ipairs(out) do print(i, k) end
```

gives the output:

```
1
 0.1420
-0.5698
[torch.DoubleTensor of size 2]

2
 0.1663
 0.1197
[torch.DoubleTensor of size 2]

3
 0.4198
-1.1394
[torch.DoubleTensor of size 2]


1
-2.4941
-1.4541
[torch.DoubleTensor of size 1x2]

2
 0.4594
 1.1946
[torch.DoubleTensor of size 1x2]

3
-2.3322
-0.7383
[torch.DoubleTensor of size 1x2]
```

A more complicated example

```
mlp = nn.Sequential()       -- Create a network that takes a Tensor
as input
mlp:add(nn.SplitTable(2))
c = nn.ParallelTable()      -- The two Tensor slices go through two
different Linear
c:add(nn.Linear(10, 3))     -- Layers in Parallel
c:add(nn.Linear(10, 7))
mlp:add(c)                  -- Outputing a table with 2 elements
p = nn.ParallelTable()      -- These tables go through two more
linear layers separately
p:add(nn.Linear(3, 2))
p:add(nn.Linear(7, 1))
mlp:add(p)
mlp:add(nn.JoinTable(1))    -- Finally, the tables are joined
together and output.

pred = mlp:forward(torch.randn(10, 2))
print(pred)

for i = 1, 100 do           -- A few steps of training such a
network..
   x = torch.ones(10, 2)
   y = torch.Tensor(3)
   y:copy(x:select(2, 1):narrow(1, 1, 3))
   pred = mlp:forward(x)

   criterion = nn.MSECriterion()
   local err = criterion:forward(pred, y)
   local gradCriterion = criterion:backward(pred, y)
   mlp:zeroGradParameters()
   mlp:backward(x, gradCriterion)
   mlp:updateParameters(0.05)

   print(err)
end
```

JoinTable

```
module = JoinTable(dimension, nInputDims)
```

Creates a module that takes a `table` of `Tensor` s as input and outputs a `Tensor` by joining them together along dimension `dimension` .
In the diagram below `dimension` is set to `1` .

```
+----------+             +-----------+
| {input1, +-------------> output[1] |
|          |           +-----------+-+
|  input2, +-----------> output[2] |
|          |         +-----------+-+
|  input3} +---------> output[3] |
+----------+         +-----------+
```

The optional parameter `nInputDims` allows to specify the number of dimensions that this module will receive. This makes it possible to forward both minibatch and non-minibatch `Tensor` s through the same module.

Example 1

```
x = torch.randn(5, 1)
y = torch.randn(5, 1)
z = torch.randn(2, 1)

print(nn.JoinTable(1):forward{x, y})
print(nn.JoinTable(2):forward{x, y})
print(nn.JoinTable(1):forward{x, z})
```

gives the output:

```
 1.3965
 0.5146
-1.5244
-0.9540
 0.4256
 0.1575
 0.4491
 0.6580
 0.1784
-1.7362
[torch.DoubleTensor of dimension 10x1]
```

```
 1.3965  0.1575
 0.5146  0.4491
-1.5244  0.6580
-0.9540  0.1784
 0.4256 -1.7362
[torch.DoubleTensor of dimension 5x2]

 1.3965
 0.5146
-1.5244
-0.9540
 0.4256
-1.2660
 1.0869
[torch.Tensor of dimension 7x1]
```

Example 2

```
module = nn.JoinTable(2, 2)

x = torch.randn(3, 1)
y = torch.randn(3, 1)

mx = torch.randn(2, 3, 1)
my = torch.randn(2, 3, 1)

print(module:forward{x, y})
print(module:forward{mx, my})
```

gives the output:

```
 0.4288  1.2002
-1.4084 -0.7960
-0.2091  0.1852
[torch.DoubleTensor of dimension 3x2]

(1,.,.) =
  0.5561  0.1228
 -0.6792  0.1153
  0.0687  0.2955
```

```
(2,.,.) =
  2.5787  1.8185
 -0.9860  0.6756
  0.1989 -0.4327
[torch.DoubleTensor of dimension 2x3x2]
```

A more complicated example

```
mlp = nn.Sequential()         -- Create a network that takes a
Tensor as input
c = nn.ConcatTable()          -- The same Tensor goes through two
different Linear
c:add(nn.Linear(10, 3))       -- Layers in Parallel
c:add(nn.Linear(10, 7))
mlp:add(c)                    -- Outputing a table with 2 elements
p = nn.ParallelTable()        -- These tables go through two more
linear layers
p:add(nn.Linear(3, 2))        -- separately.
p:add(nn.Linear(7, 1))
mlp:add(p)
mlp:add(nn.JoinTable(1))      -- Finally, the tables are joined
together and output.

pred = mlp:forward(torch.randn(10))
print(pred)

for i = 1, 100 do             -- A few steps of training such a
network..
   x = torch.ones(10)
   y = torch.Tensor(3); y:copy(x:narrow(1, 1, 3))
   pred = mlp:forward(x)

   criterion= nn.MSECriterion()
   local err = criterion:forward(pred, y)
   local gradCriterion = criterion:backward(pred, y)
   mlp:zeroGradParameters()
   mlp:backward(x, gradCriterion)
   mlp:updateParameters(0.05)

   print(err)
```

```
end
```

MixtureTable

`module` = `MixtureTable([dim])`

Creates a module that takes a `table` `{gater, experts}` as input and outputs the mixture of `experts` (a `Tensor` or `table` of `Tensor`s) using a `gater` `Tensor`. When `dim` is provided, it specifies the dimension of the `experts` `Tensor` that will be interpolated (or mixed). Otherwise, the `experts` should take the form of a `table` of `Tensor`s. This Module works for `experts` of dimension 1D or more, and for a 1D or 2D `gater`, i.e. for single examples or mini-batches.

Considering an `input = {G, E}` with a single example, then the mixture of experts `Tensor` `E` with gater `Tensor` `G` has the following form:

```
output = G[1]*E[1] + G[2]*E[2] + ... + G[n]*E[n]
```

where `dim = 1`, `n = E:size(dim) = G:size(dim)` and `G:dim() == 1`. Note that `E:dim() >= 2`, such that `output:dim() = E:dim() - 1`.

Example 1:
Using this Module, an arbitrary mixture of `n` 2-layer experts by a 2-layer gater could be constructed as follows:

```
experts = nn.ConcatTable()
for i = 1, n do
   local expert = nn.Sequential()
   expert:add(nn.Linear(3, 4))
   expert:add(nn.Tanh())
   expert:add(nn.Linear(4, 5))
   expert:add(nn.Tanh())
   experts:add(expert)
end

gater = nn.Sequential()
gater:add(nn.Linear(3, 7))
```

```
gater:add(nn.Tanh())
gater:add(nn.Linear(7, n))
gater:add(nn.SoftMax())

trunk = nn.ConcatTable()
trunk:add(gater)
trunk:add(experts)

moe = nn.Sequential()
moe:add(trunk)
moe:add(nn.MixtureTable())
```

Forwarding a batch of 2 examples gives us something like this:

```
> =moe:forward(torch.randn(2, 3))
-0.2152  0.3141  0.3280 -0.3772  0.2284
 0.2568  0.3511  0.0973 -0.0912 -0.0599
[torch.DoubleTensor of dimension 2x5]
```

Example 2:
In the following, the `MixtureTable` expects `experts` to be a `Tensor` of `size = {1, 4, 2, 5, n}`:

```
experts = nn.Concat(5)
for i = 1, n do
   local expert = nn.Sequential()
   expert:add(nn.Linear(3, 4))
   expert:add(nn.Tanh())
   expert:add(nn.Linear(4, 4*2*5))
   expert:add(nn.Tanh())
   expert:add(nn.Reshape(4, 2, 5, 1))
   experts:add(expert)
end

gater = nn.Sequential()
gater:add(nn.Linear(3, 7))
gater:add(nn.Tanh())
gater:add(nn.Linear(7, n))
gater:add(nn.SoftMax())

trunk = nn.ConcatTable()
trunk:add(gater)
trunk:add(experts)
```

```
moe = nn.Sequential()
moe:add(trunk)
moe:add(nn.MixtureTable(5))
```

Forwarding a batch of 2 examples gives us something like this:

```
> =moe:forward(torch.randn(2, 3)):size()
 2
 4
 2
 5
[torch.LongStorage of size 4]
```

SelectTable

`module` = `SelectTable(index)`

Creates a module that takes a (nested) `table` as input and outputs the element at index `index`. `index` can be strings or integers (positive or negative).
This can be either a `table` or a `Tensor`.

The gradients of the non- `index` elements are zeroed `Tensor`s of the same size. This is true regardless of the
depth of the encapsulated `Tensor` as the function used internally to do so is recursive.

Example 1:

```
> input = {torch.randn(2, 3), torch.randn(2, 1)}
> =nn.SelectTable(1):forward(input)
-0.3060  0.1398  0.2707
 0.0576  1.5455  0.0610
[torch.DoubleTensor of dimension 2x3]

> =nn.SelectTable(-1):forward(input)
 2.3080
-0.2955
[torch.DoubleTensor of dimension 2x1]
```

```
> =table.unpack(nn.SelectTable(1):backward(input, torch.randn(2,
3)))
-0.4891 -0.3495 -0.3182
-2.0999  0.7381 -0.5312
[torch.DoubleTensor of dimension 2x3]

0
0
[torch.DoubleTensor of dimension 2x1]
```

Exmaple 2:

```
> input = { A=torch.randn(2, 3), B=torch.randn(2, 1) }
> =nn.SelectTable("A"):forward(input)
-0.3060  0.1398  0.2707
 0.0576  1.5455  0.0610
[torch.DoubleTensor of dimension 2x3]

> gradInput = nn.SelectTable("A"):backward(input, torch.randn(2,
3))

> gradInput
{
  A : DoubleTensor - size: 2x3
  B : DoubleTensor - size: 2x1
}

> gradInput["A"]
-0.4891 -0.3495 -0.3182
-2.0999  0.7381 -0.5312
[torch.DoubleTensor of dimension 2x3]

> gradInput["B"]
0
0
[torch.DoubleTensor of dimension 2x1]
```

Example 3:

```
> input = {torch.randn(2, 3), {torch.randn(2, 1), {torch.randn(2,
2)}}}

> =nn.SelectTable(2):forward(input)
```

```
{
  1 : DoubleTensor - size: 2x1
  2 :
    {
      1 : DoubleTensor - size: 2x2
    }
}

> =table.unpack(nn.SelectTable(2):backward(input, {torch.randn(2,
1), {torch.randn(2, 2)}}))
0 0 0
0 0 0
[torch.DoubleTensor of dimension 2x3]

{
  1 : DoubleTensor - size: 2x1
  2 :
    {
      1 : DoubleTensor - size: 2x2
    }
}

> gradInput = nn.SelectTable(1):backward(input, torch.randn(2, 3))

> =gradInput
{
  1 : DoubleTensor - size: 2x3
  2 :
    {
      1 : DoubleTensor - size: 2x1
      2 :
        {
          1 : DoubleTensor - size: 2x2
        }
    }
}

> =gradInput[1]
-0.3400 -0.0404  1.1885
 1.2865  0.4107  0.6506
[torch.DoubleTensor of dimension 2x3]

> gradInput[2][1]
0
0
```

```
[torch.DoubleTensor of dimension 2x1]

> gradInput[2][2][1]
0 0
0 0
[torch.DoubleTensor of dimension 2x2]
```

NarrowTable

`module` = `NarrowTable(offset [, length])`

Creates a module that takes a `table` as input and outputs the subtable starting at index `offset` having `length` elements (defaults to 1 element).
The elements can be either a `table` or a `Tensor`.

The gradients of the elements not included in the subtable are zeroed `Tensor`s of the same size.
This is true regardless of the depth of the encapsulated `Tensor` as the function used internally to do so is recursive.

Example:

```
> input = {torch.randn(2, 3), torch.randn(2, 1), torch.randn(1, 2)}
> =nn.NarrowTable(2,2):forward(input)
{
  1 : DoubleTensor - size: 2x1
  2 : DoubleTensor - size: 1x2
}

> =nn.NarrowTable(1):forward(input)
{
  1 : DoubleTensor - size: 2x3
}

> =table.unpack(nn.NarrowTable(1,2):backward(input, {torch.randn(2,
3), torch.randn(2, 1)}))
 1.9528 -0.1381  0.2023
 0.2297 -1.5169 -1.1871
[torch.DoubleTensor of size 2x3]
```

```
-1.2023
-0.4165
[torch.DoubleTensor of size 2x1]

 0  0
[torch.DoubleTensor of size 1x2]
```

FlattenTable

`module` = `FlattenTable()`

Creates a module that takes an arbitrarily deep `table` of `Tensor`s (potentially nested) as input and outputs a `table` of `Tensor`s, where the output `Tensor` in index `i` is the `Tensor` with post-order DFS index `i` in the input `table`.

This module is particularly useful in combination with nn.Identity() to create networks that can append to their input `table`.

Example:

```
x = {torch.rand(1), {torch.rand(2), {torch.rand(3)}},
torch.rand(4)}
print(x)
print(nn.FlattenTable():forward(x))
```

gives the output:

```
{
  1 : DoubleTensor - size: 1
  2 :
    {
      1 : DoubleTensor - size: 2
      2 :
        {
          1 : DoubleTensor - size: 3
        }
    }
  3 : DoubleTensor - size: 4
}
```

```
{
  1 : DoubleTensor - size: 1
  2 : DoubleTensor - size: 2
  3 : DoubleTensor - size: 3
  4 : DoubleTensor - size: 4
}
```

PairwiseDistance

`module` = `PairwiseDistance(p)` creates a module that takes a `table` of two vectors as input and outputs the distance between them using the `p`-norm.

Example:

```
mlp_l1 = nn.PairwiseDistance(1)
mlp_l2 = nn.PairwiseDistance(2)
x = torch.Tensor({1, 2, 3})
y = torch.Tensor({4, 5, 6})
print(mlp_l1:forward({x, y}))
print(mlp_l2:forward({x, y}))
```

gives the output:

```
 9
[torch.Tensor of dimension 1]

 5.1962
[torch.Tensor of dimension 1]
```

A more complicated example:

```
-- imagine we have one network we are interested in, it is called
"p1_mlp"
p1_mlp= nn.Sequential(); p1_mlp:add(nn.Linear(5, 2))

-- But we want to push examples towards or away from each other
-- so we make another copy of it called p2_mlp
-- this *shares* the same weights via the set command, but has its
```

```
own set of temporary gradient storage
-- that's why we create it again (so that the gradients of the pair
don't wipe each other)
p2_mlp= nn.Sequential(); p2_mlp:add(nn.Linear(5, 2))
p2_mlp:get(1).weight:set(p1_mlp:get(1).weight)
p2_mlp:get(1).bias:set(p1_mlp:get(1).bias)

-- we make a parallel table that takes a pair of examples as input.
they both go through the same (cloned) mlp
prl = nn.ParallelTable()
prl:add(p1_mlp)
prl:add(p2_mlp)

-- now we define our top level network that takes this parallel
table and computes the pairwise distance between
-- the pair of outputs
mlp= nn.Sequential()
mlp:add(prl)
mlp:add(nn.PairwiseDistance(1))

-- and a criterion for pushing together or pulling apart pairs
crit = nn.HingeEmbeddingCriterion(1)

-- lets make two example vectors
x = torch.rand(5)
y = torch.rand(5)


-- Use a typical generic gradient update function
function gradUpdate(mlp, x, y, criterion, learningRate)
local pred = mlp:forward(x)
local err = criterion:forward(pred, y)
local gradCriterion = criterion:backward(pred, y)
mlp:zeroGradParameters()
mlp:backward(x, gradCriterion)
mlp:updateParameters(learningRate)
end

-- push the pair x and y together, notice how then the distance
between them given
-- by  print(mlp:forward({x, y})[1]) gets smaller
for i = 1, 10 do
gradUpdate(mlp, {x, y}, 1, crit, 0.01)
print(mlp:forward({x, y})[1])
end
```

```
-- pull apart the pair x and y, notice how then the distance
between them given
-- by  print(mlp:forward({x, y})[1]) gets larger

for i = 1, 10 do
gradUpdate(mlp, {x, y}, -1, crit, 0.01)
print(mlp:forward({x, y})[1])
end
```

DotProduct

`module` = `DotProduct()` creates a module that takes a `table` of two vectors (or matrices if in batch mode) as input and outputs the dot product between them.

Example:

```
mlp = nn.DotProduct()
x = torch.Tensor({1, 2, 3})
y = torch.Tensor({4, 5, 6})
print(mlp:forward({x, y}))
```

gives the output:

```
 32
[torch.Tensor of dimension 1]
```

A more complicated example:

```
-- Train a ranking function so that mlp:forward({x, y}, {x, z})
returns a number
-- which indicates whether x is better matched with y or z (larger
score = better match), or vice versa.

mlp1 = nn.Linear(5, 10)
mlp2 = mlp1:clone('weight', 'bias')
```

```
prl = nn.ParallelTable();
prl:add(mlp1); prl:add(mlp2)

mlp1 = nn.Sequential()
mlp1:add(prl)
mlp1:add(nn.DotProduct())

mlp2 = mlp1:clone('weight', 'bias')

mlp = nn.Sequential()
prla = nn.ParallelTable()
prla:add(mlp1)
prla:add(mlp2)
mlp:add(prla)

x = torch.rand(5);
y = torch.rand(5)
z = torch.rand(5)


print(mlp1:forward{x, x})
print(mlp1:forward{x, y})
print(mlp1:forward{y, y})


crit = nn.MarginRankingCriterion(1);

-- Use a typical generic gradient update function
function gradUpdate(mlp, x, y, criterion, learningRate)
   local pred = mlp:forward(x)
   local err = criterion:forward(pred, y)
   local gradCriterion = criterion:backward(pred, y)
   mlp:zeroGradParameters()
   mlp:backward(x, gradCriterion)
   mlp:updateParameters(learningRate)
end

inp = {{x, y}, {x, z}}

math.randomseed(1)

-- make the pair x and y have a larger dot product than x and z

for i = 1, 100 do
```

```
   gradUpdate(mlp, inp, 1, crit, 0.05)
   o1 = mlp1:forward{x, y}[1];
   o2 = mlp2:forward{x, z}[1];
   o = crit:forward(mlp:forward{{x, y}, {x, z}}, 1)
   print(o1, o2, o)
end

print "_________________**"

-- make the pair x and z have a larger dot product than x and y

for i = 1, 100 do
   gradUpdate(mlp, inp, -1, crit, 0.05)
   o1 = mlp1:forward{x, y}[1];
   o2 = mlp2:forward{x, z}[1];
   o = crit:forward(mlp:forward{{x, y}, {x, z}}, -1)
   print(o1, o2, o)
end
```

CosineDistance

`module` = `CosineDistance()` creates a module that takes a `table` of two vectors (or matrices if in batch mode) as input and outputs the cosine distance between them.

Examples:

```
mlp = nn.CosineDistance()
x = torch.Tensor({1, 2, 3})
y = torch.Tensor({4, 5, 6})
print(mlp:forward({x, y}))
```

gives the output:

```
 0.9746
[torch.Tensor of dimension 1]
```

`CosineDistance` also accepts batches:

```
mlp = nn.CosineDistance()
x = torch.Tensor({{1,2,3},{1,2,-3}})
y = torch.Tensor({{4,5,6},{-4,5,6}})
print(mlp:forward({x,y}))
```

gives the output:

```
 0.9746
-0.3655
[torch.DoubleTensor of size 2]
```

A more complicated example:

```
-- imagine we have one network we are interested in, it is called
"p1_mlp"
p1_mlp= nn.Sequential(); p1_mlp:add(nn.Linear(5, 2))

-- But we want to push examples towards or away from each other
-- so we make another copy of it called p2_mlp
-- this *shares* the same weights via the set command, but has its
own set of temporary gradient storage
-- that's why we create it again (so that the gradients of the pair
don't wipe each other)
p2_mlp= p1_mlp:clone('weight', 'bias')

-- we make a parallel table that takes a pair of examples as input.
they both go through the same (cloned) mlp
prl = nn.ParallelTable()
prl:add(p1_mlp)
prl:add(p2_mlp)

-- now we define our top level network that takes this parallel
table and computes the cosine distance between
-- the pair of outputs
mlp= nn.Sequential()
mlp:add(prl)
mlp:add(nn.CosineDistance())


-- lets make two example vectors
x = torch.rand(5)
```

```
y = torch.rand(5)

-- Grad update function..
function gradUpdate(mlp, x, y, learningRate)
    local pred = mlp:forward(x)
    if pred[1]*y < 1 then
        gradCriterion = torch.Tensor({-y})
        mlp:zeroGradParameters()
        mlp:backward(x, gradCriterion)
        mlp:updateParameters(learningRate)
    end
end

-- push the pair x and y together, the distance should get larger..
for i = 1, 1000 do
 gradUpdate(mlp, {x, y}, 1, 0.1)
 if ((i%100)==0) then print(mlp:forward({x, y})[1]);end
end


-- pull apart the pair x and y, the distance should get smaller..

for i = 1, 1000 do
 gradUpdate(mlp, {x, y}, -1, 0.1)
 if ((i%100)==0) then print(mlp:forward({x, y})[1]);end
end
```

CriterionTable

`module` = `CriterionTable(criterion)`

Creates a module that wraps a Criterion module so that it can accept a `table` of inputs. Typically the `table` would contain two elements: the input and output `x` and `y` that the Criterion compares.

Example:

```
mlp = nn.CriterionTable(nn.MSECriterion())
x = torch.randn(5)
y = torch.randn(5)
```

```
print(mlp:forward{x, x})
print(mlp:forward{x, y})
```

gives the output:

```
0
1.9028918413199
```

Here is a more complex example of embedding the criterion into a network:

```
function table.print(t)
 for i, k in pairs(t) do print(i, k); end
end

mlp = nn.Sequential();                          -- Create an mlp
that takes input
  main_mlp = nn.Sequential();              -- and output using
ParallelTable
  main_mlp:add(nn.Linear(5, 4))
  main_mlp:add(nn.Linear(4, 3))
 cmlp = nn.ParallelTable();
 cmlp:add(main_mlp)
 cmlp:add(nn.Identity())
mlp:add(cmlp)
mlp:add(nn.CriterionTable(nn.MSECriterion())) -- Apply the
Criterion

for i = 1, 20 do                                -- Train for a few
iterations
 x = torch.ones(5);
 y = torch.Tensor(3); y:copy(x:narrow(1, 1, 3))
 err = mlp:forward{x, y}                        -- Pass in both
input and output
 print(err)

 mlp:zeroGradParameters();
 mlp:backward({x, y} );
 mlp:updateParameters(0.05);
end
```

CAddTable

`module` = `CAddTable([inplace])`

Takes a `table` of `Tensor`s and outputs summation of all `Tensor`s. If `inplace` is `true`, the sum is written to the first `Tensor`.

```
ii = {torch.ones(5), torch.ones(5)*2, torch.ones(5)*3}
=ii[1]
 1
 1
 1
 1
 1
[torch.DoubleTensor of dimension 5]

return ii[2]
 2
 2
 2
 2
 2
[torch.DoubleTensor of dimension 5]

return ii[3]
 3
 3
 3
 3
 3
[torch.DoubleTensor of dimension 5]

m = nn.CAddTable()
=m:forward(ii)
 6
 6
 6
 6
 6
[torch.DoubleTensor of dimension 5]
```

CSubTable

Takes a `table` with two `Tensor` and returns the component-wise subtraction between them.

```
m = nn.CSubTable()
=m:forward({torch.ones(5)*2.2, torch.ones(5)})
 1.2000
 1.2000
 1.2000
 1.2000
 1.2000
[torch.DoubleTensor of dimension 5]
```

CMulTable

Takes a `table` of `Tensor`s and outputs the multiplication of all of them.

```
ii = {torch.ones(5)*2, torch.ones(5)*3, torch.ones(5)*4}
m = nn.CMulTable()
=m:forward(ii)
 24
 24
 24
 24
 24
[torch.DoubleTensor of dimension 5]
```

CDivTable

Takes a `table` with two `Tensor` and returns the component-wise division between them.

```
m = nn.CDivTable()
=m:forward({torch.ones(5)*2.2, torch.ones(5)*4.4})
 0.5000
 0.5000
 0.5000
 0.5000
 0.5000
[torch.DoubleTensor of dimension 5]
```

CMaxTable

Takes a `table` of `Tensor`s and outputs the max of all of them.

```
m = nn.CMaxTable()
=m:forward({{torch.Tensor{1,2,3}, torch.Tensor{3,2,1}})
 3
 2
 3
[torch.DoubleTensor of size 3]
```

CMinTable

Takes a `table` of `Tensor`s and outputs the min of all of them.

```
m = nn.CMinTable()
=m:forward({{torch.Tensor{1,2,3}, torch.Tensor{3,2,1}})
 1
 2
 1
[torch.DoubleTensor of size 3]
```

Criterions

`Criterions` are helpful to train a neural network. Given an input and a target, they compute a gradient according to a given loss function.

- Classification criterions:
  - `BCECriterion` : binary cross-entropy for `Sigmoid` (two-class version of `ClassNLLCriterion`);
  - `ClassNLLCriterion` : negative log-likelihood for `LogSoftMax` (multi-class);
  - `CrossEntropyCriterion` : combines `LogSoftMax` and `ClassNLLCriterion`;
  - `ClassSimplexCriterion` : A simplex embedding criterion for classification.
  - `MarginCriterion` : two class margin-based loss;
  - `SoftMarginCriterion` : two class softmargin-based loss;
  - `MultiMarginCriterion` : multi-class margin-based loss;
  - `MultiLabelMarginCriterion` : multi-class multi-classification margin-based loss;
  - `MultiLabelSoftMarginCriterion` : multi-class multi-classification loss based on binary cross-entropy;

- Regression criterions:
  - `AbsCriterion` : measures the mean absolute value of the element-wise difference between input;
  - `SmoothL1Criterion` : a smooth version of the AbsCriterion;
  - `MSECriterion` : mean square error (a classic);
  - `SpatialAutoCropMSECriterion` : Spatial mean square error when the input is spatially smaller than the target, by only comparing their spatial overlap;
  - `DistKLDivCriterion` : Kullback–Leibler divergence (for fitting continuous probability distributions);

- Embedding criterions (measuring whether two inputs are similar or dissimilar):
  - `HingeEmbeddingCriterion` : takes a distance as input;
  - `L1HingeEmbeddingCriterion` : L1 distance between two inputs;
  - `CosineEmbeddingCriterion` : cosine distance between two inputs;
  - `DistanceRatioCriterion` : Probabilistic criterion for training siamese model with triplets.

- Miscelaneus criterions:
  - `MultiCriterion` : a weighted sum of other criterions each applied to the same input and target;
  - `ParallelCriterion` : a weighted sum of other criterions each applied to a

different input and target;
- `MarginRankingCriterion` : ranks two inputs;

Criterion

This is an abstract class which declares methods defined in all criterions.
This class is serializable.

[output] forward(input, target)

Given an `input` and a `target`, compute the loss function associated to the criterion and return the result.
In general `input` and `target` are `Tensor`s, but some specific criterions might require some other type of object.

The `output` returned should be a scalar in general.

The state variable `self.output` should be updated after a call to `forward()`.

[gradInput] backward(input, target)

Given an `input` and a `target`, compute the gradients of the loss function associated to the criterion and return the result.
In general `input`, `target` and `gradInput` are `Tensor`s, but some specific criterions might require some other type of object.

The state variable `self.gradInput` should be updated after a call to `backward()`.

State variable: output

State variable which contains the result of the last `forward(input, target)` call.

State variable: gradInput

State variable which contains the result of the last `backward(input, target)` call.

AbsCriterion

```
criterion = nn.AbsCriterion()
```

Creates a criterion that measures the mean absolute value of the element-wise difference between input `x` and target `y`:

```
loss(x, y)  = 1/n \sum |x_i - y_i|
```

If `x` and `y` are `d`-dimensional `Tensor`s with a total of `n` elements, the sum operation still operates over all the elements, and divides by `n`.

The division by `n` can be avoided if one sets the internal variable `sizeAverage` to `false`:

```
criterion = nn.AbsCriterion()
criterion.sizeAverage = false
```

ClassNLLCriterion

```
criterion = nn.ClassNLLCriterion([weights])
```

The negative log likelihood criterion. It is useful to train a classification problem with `n` classes.
If provided, the optional argument `weights` should be a 1D `Tensor` assigning weight to each of the classes.
This is particularly useful when you have an unbalanced training set.

The `input` given through a `forward()` is expected to contain *log-probabilities* of each class: `input` has to be a 1D `Tensor` of size `n`.
Obtaining log-probabilities in a neural network is easily achieved by adding a `LogSoftMax` layer in the last layer of your neural network.

You may use `CrossEntropyCriterion` instead, if you prefer not to add an extra layer to your network.
This criterion expects a class index (1 to the number of class) as `target` when calling `forward(input, target)` and `backward(input, target)`.

The loss can be described as:

```
loss(x, class) = -x[class]
```

or in the case of the `weights` argument it is specified as follows:

```
loss(x, class) = -weights[class] * x[class]
```

Due to the behaviour of the backend code, it is necessary to set sizeAverage to false when calculating losses *in non-batch mode*.

The following is a code fragment showing how to make a gradient step given an input `x`, a desired output `y` (an integer `1` to `n`, in this case `n = 2` classes), a network `mlp` and a learning rate `learningRate`:

```
function gradUpdate(mlp, x, y, learningRate)
   local criterion = nn.ClassNLLCriterion()
   local pred = mlp:forward(x)
   local err = criterion:forward(pred, y)
   mlp:zeroGradParameters()
   local t = criterion:backward(pred, y)
   mlp:backward(x, t)
   mlp:updateParameters(learningRate)
end
```

By default, the losses are averaged over observations for each minibatch. However, if the field `sizeAverage` is set to `false`, the losses are instead summed for each minibatch.

CrossEntropyCriterion

```
criterion = nn.CrossEntropyCriterion([weights])
```

This criterion combines `LogSoftMax` and `ClassNLLCriterion` in one single class.

It is useful to train a classification problem with `n` classes.
If provided, the optional argument `weights` should be a 1D `Tensor` assigning weight to each of the classes. This is particularly useful when you have an unbalanced training set.

The `input` given through a `forward()` is expected to contain scores for each class: `input` has to be a 1D `Tensor` of size `n`.
This criterion expect a class index (1 to the number of class) as `target` when calling `forward(input, target)` and `backward(input, target)`.

The loss can be described as:

```
loss(x, class) = -log(exp(x[class]) / (\sum_j exp(x[j])))
               = -x[class] + log(\sum_j exp(x[j]))
```

or in the case of the `weights` argument being specified:

```
loss(x, class) = weights[class] * (-x[class] + log(\sum_j
exp(x[j])))
```

Due to the behaviour of the backend code, it is necessary to set sizeAverage to false when calculating losses *in non-batch mode*.

```
crit = nn.CrossEntropyCriterion(weights)
crit.nll.sizeAverage = false
```

The losses are averaged across observations for each minibatch.

ClassSimplexCriterion

```
criterion = nn.ClassSimplexCriterion(nClasses)
```

ClassSimplexCriterion implements a criterion for classification.
It learns an embedding per class, where each class' embedding is a point on an (N-1)-dimensional simplex,
where N is the number of classes.

The `input` given through a `forward()` is expected to be the output of a Normalized Linear layer with no bias:
- `input` has to be a 1D `Tensor` of size `n` for a single sample
- a 2D `Tensor` of size `batchSize x n` for a mini-batch of samples

This Criterion is best used in combination with a neural network where the last layers are:
- a weight-normalized bias-less Linear layer. Example source code
- followed by an output normalization layer (nn.Normalize).

The loss is described in detail in the paper Scale-invariant learning and convolutional networks.

The following is a code fragment showing how to make a gradient step given an input `x`, a desired output `y` (an integer `1` to `n`, in this case `n = 30` classes), a network `mlp` and a learning rate `learningRate`:

```
nInput = 10
nClasses = 30
nHidden = 100
mlp = nn.Sequential()
mlp:add(nn.Linear(nInput, nHidden)):add(nn.ReLU())
mlp:add(nn.NormalizedLinearNoBias(nHidden, nClasses))
mlp:add(nn.Normalize(2))

criterion = nn.ClassSimplexCriterion(nClasses)

function gradUpdate(mlp, x, y, learningRate)
   pred = mlp:forward(x)
   local err = criterion:forward(pred, y)
   mlp:zeroGradParameters()
   local t = criterion:backward(pred, y)
   mlp:backward(x, t)
   mlp:updateParameters(learningRate)
end
```

This criterion also provides two helper functions `getPredictions(input)` and `getTopPrediction(input)` that return the raw predictions and the top prediction index respectively, given an input sample.

DistKLDivCriterion

```
criterion = nn.DistKLDivCriterion()
```

The Kullback–Leibler divergence criterion.
KL divergence is a useful distance measure for continuous distributions and is often useful when performing direct regression over the space of (discretely sampled) continuous output distributions.
As with ClassNLLCriterion, the `input` given through a `forward()` is expected to contain *log-probabilities*, however unlike ClassNLLCriterion, `input` is not restricted to a 1D or 2D vector (as the criterion is applied element-wise).

This criterion expect a `target` `Tensor` of the same size as the `input` `Tensor` when calling `forward(input, target)` and `backward(input, target)`.

The loss can be described as:

```
loss(x, target) = 1/n \sum(target_i * (log(target_i) - x_i))
```

By default, the losses are averaged for each minibatch over observations *as well as* over dimensions. However, if the field `sizeAverage` is set to `false`, the losses are instead summed.

BCECriterion

```
criterion = nn.BCECriterion([weights])
```

Creates a criterion that measures the Binary Cross Entropy between the target and the output:

```
loss(o, t) = - 1/n sum_i (t[i] * log(o[i]) + (1 - t[i]) * log(1 -
o[i]))
```

or in the case of the weights argument being specified:

```
loss(o, t) = - 1/n sum_i weights[i] * (t[i] * log(o[i]) + (1 -
t[i]) * log(1 - o[i]))
```

This is used for measuring the error of a reconstruction in for example an auto-encoder. Note that the outputs `o[i]` should be numbers between 0 and 1, for instance, the output of an `nn.Sigmoid` layer and should be interpreted as the probability of predicting `t[i] = 1`. Note `t[i]` can be either 0 or 1.

By default, the losses are averaged for each minibatch over observations *as well as* over dimensions. However, if the field `sizeAverage` is set to `false`, the losses are instead summed.

MarginCriterion

```
criterion = nn.MarginCriterion([margin])
```

Creates a criterion that optimizes a two-class classification hinge loss (margin-based loss) between input `x` (a `Tensor` of dimension `1`) and output `y` (which is a tensor containing either `1` s or `-1` s).
`margin`, if unspecified, is by default `1`.

```
loss(x, y) = sum_i (max(0, margin - y[i]*x[i])) / x:nElement()
```

The normalization by the number of elements in the input can be disabled by setting `self.sizeAverage` to `false`.

Example

```
function gradUpdate(mlp, x, y, criterion, learningRate)
   local pred = mlp:forward(x)
   local err = criterion:forward(pred, y)
   local gradCriterion = criterion:backward(pred, y)
   mlp:zeroGradParameters()
   mlp:backward(x, gradCriterion)
   mlp:updateParameters(learningRate)
end

mlp = nn.Sequential()
mlp:add(nn.Linear(5, 1))
```

```
x1 = torch.rand(5)
x1_target = torch.Tensor{1}
x2 = torch.rand(5)
x2_target = torch.Tensor{-1}
criterion=nn.MarginCriterion(1)

for i = 1, 1000 do
   gradUpdate(mlp, x1, x1_target, criterion, 0.01)
   gradUpdate(mlp, x2, x2_target, criterion, 0.01)
end

print(mlp:forward(x1))
print(mlp:forward(x2))

print(criterion:forward(mlp:forward(x1), x1_target))
print(criterion:forward(mlp:forward(x2), x2_target))
```

gives the output:

```
 1.0043
[torch.Tensor of dimension 1]


-1.0061
[torch.Tensor of dimension 1]

0
0
```

i.e. the mlp successfully separates the two data points such that they both have a `margin` of `1`, and hence a loss of `0`.

By default, the losses are averaged over observations for each minibatch. However, if the field `sizeAverage` is set to `false`, the losses are instead summed.

SoftMarginCriterion

```
criterion = nn.SoftMarginCriterion()
```

Creates a criterion that optimizes a two-class classification logistic loss between input `x` (a `Tensor` of dimension `1`) and output `y` (which is a tensor containing either `1`s or `-1`s).

```
loss(x, y) = sum_i (log(1 + exp(-y[i]*x[i]))) / x:nElement()
```

The normalization by the number of elements in the input can be disabled by setting `self.sizeAverage` to `false`.

Example

```
function gradUpdate(mlp, x, y, criterion, learningRate)
   local pred = mlp:forward(x)
   local err = criterion:forward(pred, y)
   local gradCriterion = criterion:backward(pred, y)
   mlp:zeroGradParameters()
   mlp:backward(x, gradCriterion)
   mlp:updateParameters(learningRate)
end

mlp = nn.Sequential()
mlp:add(nn.Linear(5, 1))

x1 = torch.rand(5)
x1_target = torch.Tensor{1}
x2 = torch.rand(5)
x2_target = torch.Tensor{-1}
criterion=nn.SoftMarginCriterion(1)

for i = 1, 1000 do
   gradUpdate(mlp, x1, x1_target, criterion, 0.01)
   gradUpdate(mlp, x2, x2_target, criterion, 0.01)
end

print(mlp:forward(x1))
print(mlp:forward(x2))

print(criterion:forward(mlp:forward(x1), x1_target))
print(criterion:forward(mlp:forward(x2), x2_target))
```

gives the output:

```
 0.7471
[torch.DoubleTensor of size 1]

-0.9607
[torch.DoubleTensor of size 1]

0.38781049558836
0.32399356957564
```

i.e. the mlp successfully separates the two data points.

By default, the losses are averaged over observations for each minibatch. However, if the field `sizeAverage` is set to `false`, the losses are instead summed.

MultiMarginCriterion

```
criterion = nn.MultiMarginCriterion(p, [weights], [margin])
```

Creates a criterion that optimizes a multi-class classification hinge loss (margin-based loss) between input `x` (a `Tensor` of dimension 1) and output `y` (which is a target class index, `1` <= `y` <= `x:size(1)`):

```
loss(x, y) = sum_i(max(0, (margin - x[y] + x[i]))^p) / x:size(1)
```

where `i == 1` to `x:size(1)` and `i ~= y`.
Note that this criterion also works with 2D inputs and 1D targets.

Optionally, you can give non-equal weighting on the classes by passing a 1D `weights` tensor into the constructor.
The loss function then becomes:

```
loss(x, y) = sum_i(max(0, w[y] * (margin - x[y] - x[i]))^p) /
x:size(1)
```

This criterion is especially useful for classification when used in conjunction with a module ending in the following output layer:

```
mlp = nn.Sequential()
mlp:add(nn.Euclidean(n, m)) -- outputs a vector of distances
mlp:add(nn.MulConstant(-1)) -- distance to similarity
```

By default, the losses are averaged over observations for each minibatch. However, if the field `sizeAverage` is set to `false`, the losses are instead summed.

MultiLabelMarginCriterion

```
criterion = nn.MultiLabelMarginCriterion()
```

Creates a criterion that optimizes a multi-class multi-classification hinge loss (margin-based loss) between input `x` (a 1D `Tensor`) and output `y` (which is a 1D `Tensor` of target class indices):

```
loss(x, y) = sum_ij(max(0, 1 - (x[y[j]] - x[i]))) / x:size(1)
```

where `i == 1` to `x:size(1)`, `j == 1` to `y:size(1)`, `y[j] ~= 0`, and `i ~= y[j]` for all `i` and `j`.
Note that this criterion also works with 2D inputs and targets.

`y` and `x` must have the same size.
The criterion only considers the first non zero `y[j]` targets.
This allows for different samples to have variable amounts of target classes:

```
criterion = nn.MultiLabelMarginCriterion()
input = torch.randn(2, 4)
target = torch.Tensor{{1, 3, 0, 0}, {4, 0, 0, 0}} -- zero-values
are ignored
criterion:forward(input, target)
```

MultiLabelSoftMarginCriterion

MultiLabelSoftMarginCriterion

```
criterion = nn.MultiLabelSoftMarginCriterion()
```

Creates a criterion that optimizes a multi-label one-versus-all loss based on max-entropy, between input `x` (a 1D `Tensor`) and target `y` (a binary 1D `Tensor`):

```
loss(x, y) = - sum_i (y[i] log( exp(x[i]) / (1 + exp(x[i]))) + (1-
y[i]) log(1/(1+exp(x[i])))) / x:nElement()
```

where `i == 1` to `x:nElement()`, `y[i] in {0,1}`.
Note that this criterion also works with 2D inputs and targets.

`y` and `x` must have the same size.

MSECriterion

```
criterion = nn.MSECriterion()
```

Creates a criterion that measures the mean squared error between `n` elements in the input `x` and output `y`:

```
loss(x, y) = 1/n \sum |x_i - y_i|^2 .
```

If `x` and `y` are `d`-dimensional `Tensor`s with a total of `n` elements, the sum operation still operates over all the elements, and divides by `n`.
The two `Tensor`s must have the same number of elements (but their sizes might be different).

The division by `n` can be avoided if one sets the internal variable `sizeAverage` to `false`:

```
criterion = nn.MSECriterion()
criterion.sizeAverage = false
```

By default, the losses are averaged over observations for each minibatch. However, if the field

`sizeAverage` is set to `false`, the losses are instead summed.

SpatialAutoCropMSECriterion

```
criterion = nn.SpatialAutoCropMSECriterion()
```

Creates a criterion that measures the mean squared error between the input and target, even if the target is spatially larger than the input. It achieves this by center-cropping the target to the same spatial resolution as the input, the mean squared error is then calculated between the input and this cropped target.

If the input and cropped target tensors are `d`-dimensional `Tensor`s with a total of `n` elements, the sum operation operates over all the elements, and divides by `n`.

The division by `n` can be avoided if one sets the internal variable `sizeAverage` to `false`:

```
criterion = nn.SpatialAutoCropMSECriterion()
criterion.sizeAverage = false
```

MultiCriterion

```
criterion = nn.MultiCriterion()
```

This returns a Criterion which is a weighted sum of other Criterion.
Criterions are added using the method:

```
criterion:add(singleCriterion [, weight])
```

where `weight` is a scalar (default 1). Each criterion is applied to the same `input` and `target`.

Example :

```
input = torch.rand(2,10)
target = torch.IntTensor{1,8}
nll = nn.ClassNLLCriterion()
nll2 = nn.CrossEntropyCriterion()
mc = nn.MultiCriterion():add(nll, 0.5):add(nll2)
output = mc:forward(input, target)
```

ParallelCriterion

```
criterion = nn.ParallelCriterion([repeatTarget])
```

This returns a Criterion which is a weighted sum of other Criterion.
Criterions are added using the method:

```
criterion:add(singleCriterion [, weight])
```

where `weight` is a scalar (default 1). The criterion expects an `input` and `target` table. Each criterion is applied to the commensurate `input` and `target` element in the tables. However, if `repeatTarget=true`, the `target` is repeatedly presented to each criterion (with a different `input`).

Example :

```
input = {torch.rand(2,10), torch.randn(2,10)}
target = {torch.IntTensor{1,8}, torch.randn(2,10)}
nll = nn.ClassNLLCriterion()
mse = nn.MSECriterion()
pc = nn.ParallelCriterion():add(nll, 0.5):add(mse)
output = pc:forward(input, target)
```

SmoothL1Criterion

```
criterion = nn.SmoothL1Criterion()
```

Creates a criterion that can be thought of as a smooth version of the `AbsCriterion`. It uses a squared term if the absolute element-wise error falls below 1. It is less sensitive to outliers than the `MSECriterion` and in some cases prevents exploding gradients (e.g. see “Fast R-CNN” paper by Ross Girshick).

```
                      ⎧ 0.5 * (x_i - y_i)^2, if |x_i - y_i| < 1
loss(x, y) = 1/n \sum ⎨
                      ⎩ |x_i - y_i| - 0.5,   otherwise
```

If `x` and `y` are `d`-dimensional `Tensor`s with a total of `n` elements, the sum operation still operates over all the elements, and divides by `n`.

The division by `n` can be avoided if one sets the internal variable `sizeAverage` to `false`:

```
criterion = nn.SmoothL1Criterion()
criterion.sizeAverage = false
```

By default, the losses are averaged over observations for each minibatch. However, if the field `sizeAverage` is set to `false`, the losses are instead summed.

HingeEmbeddingCriterion

```
criterion = nn.HingeEmbeddingCriterion([margin])
```

Creates a criterion that measures the loss given an input `x` which is a 1-dimensional vector and a label `y` (`1` or `-1`).
This is usually used for measuring whether two inputs are similar or dissimilar, e.g. using the L1 pairwise distance, and is typically used for learning nonlinear embeddings or semi-supervised learning.

```
                  ⎧ x_i,                  if y_i ==  1
loss(x, y) = 1/n ⎨
                  ⎩ max(0, margin - x_i), if y_i == -1
```

If `x` and `y` are `n`-dimensional `Tensor`s, the sum operation still operates over all the elements, and divides by `n` (this can be avoided if one sets the internal variable `sizeAverage` to `false`). The `margin` has a default value of `1`, or can be set in the constructor.

Example

```
-- imagine we have one network we are interested in, it is called
"p1_mlp"
p1_mlp = nn.Sequential(); p1_mlp:add(nn.Linear(5, 2))

-- But we want to push examples towards or away from each other so
we make another copy
-- of it called p2_mlp; this *shares* the same weights via the set
command, but has its
-- own set of temporary gradient storage that's why we create it
again (so that the gradients
-- of the pair don't wipe each other)
p2_mlp = nn.Sequential(); p2_mlp:add(nn.Linear(5, 2))
p2_mlp:get(1).weight:set(p1_mlp:get(1).weight)
p2_mlp:get(1).bias:set(p1_mlp:get(1).bias)

-- we make a parallel table that takes a pair of examples as input.
-- They both go through the same (cloned) mlp
prl = nn.ParallelTable()
prl:add(p1_mlp)
prl:add(p2_mlp)

-- now we define our top level network that takes this parallel
table
-- and computes the pairwise distance betweem the pair of outputs
mlp = nn.Sequential()
mlp:add(prl)
mlp:add(nn.PairwiseDistance(1))

-- and a criterion for pushing together or pulling apart pairs
crit = nn.HingeEmbeddingCriterion(1)

-- lets make two example vectors
x = torch.rand(5)
y = torch.rand(5)
```

```
-- Use a typical generic gradient update function
function gradUpdate(mlp, x, y, criterion, learningRate)
local pred = mlp:forward(x)
local err = criterion:forward(pred, y)
local gradCriterion = criterion:backward(pred, y)
mlp:zeroGradParameters()
mlp:backward(x, gradCriterion)
mlp:updateParameters(learningRate)
end

-- push the pair x and y together, notice how then the distance
between them given
-- by print(mlp:forward({x, y})[1]) gets smaller
for i = 1, 10 do
   gradUpdate(mlp, {x, y}, 1, crit, 0.01)
   print(mlp:forward({x, y})[1])
end

-- pull apart the pair x and y, notice how then the distance
between them given
-- by print(mlp:forward({x, y})[1]) gets larger

for i = 1, 10 do
   gradUpdate(mlp, {x, y}, -1, crit, 0.01)
   print(mlp:forward({x, y})[1])
end
```

By default, the losses are averaged over observations for each minibatch. However, if the field `sizeAverage` is set to `false`, the losses are instead summed.

L1HingeEmbeddingCriterion

```
criterion = nn.L1HingeEmbeddingCriterion([margin])
```

Creates a criterion that measures the loss given an input `x` = `{x1, x2}`, a table of two `Tensor`s, and a label `y` (`1` or `-1`): this is used for measuring whether two inputs are similar or dissimilar, using the L1 distance, and is typically used for learning nonlinear embeddings or semi-supervised learning.

```
             ⎧ ||x1 - x2||_1,                  if y ==  1
loss(x, y) = ⎨
             ⎩ max(0, margin - ||x1 - x2||_1), if y == -1
```

The `margin` has a default value of `1`, or can be set in the constructor.

CosineEmbeddingCriterion

```
criterion = nn.CosineEmbeddingCriterion([margin])
```

Creates a criterion that measures the loss given an input `x` = `{x1, x2}`, a table of two `Tensor`s, and a `Tensor` label `y` with values 1 or -1.
This is used for measuring whether two inputs are similar or dissimilar, using the cosine distance, and is typically used for learning nonlinear embeddings or semi-supervised learning.

`margin` should be a number from `-1` to `1`, `0` to `0.5` is suggested.
`Forward` and `Backward` have to be used alternately. If `margin` is missing, the default value is `0`.

The loss function for each sample is:

```
             ⎧ 1 - cos(x1, x2),              if y ==  1
loss(x, y) = ⎨
             ⎩ max(0, cos(x1, x2) - margin), if y == -1
```

For batched inputs, if the internal variable `sizeAverage` is equal to `true`, the loss function averages the loss over the batch samples; if `sizeAverage` is `false`, then the loss function sums over the batch samples. By default, `sizeAverage` equals to `true`.

By default, the losses are averaged over observations for each minibatch. However, if the field `sizeAverage` is set to `false`, the losses are instead summed.

DistanceRatioCriterion

Ref A. Unsupervised Learning through Spatial Contrasting

```
criterion = nn.DistanceRatioCriterion(sizeAverage)
```

This criterion is probabilistic treatment of margin cost. The model is trained using sample triplets `{Xs, Xa, Xd}` where `Xa` is anchor sample, `Xs` is sample similar to anchor sample and `Xd` is a sample not similar to anchor sample. Let `Ds` be distance between embeddings of `{Xs, Xa}` and `Dd` be distance between embeddings of `{Xa, Xd}` then the loss is defined as follow

```
    loss = -log( exp(-Ds) / ( exp(-Ds) + exp(-Dd) ) )
```

Sample example

```
    torch.setdefaulttensortype("torch.FloatTensor")

    require 'nn'

    -- triplet : with batchSize of 32 and dimensionality 512
    sample = {torch.rand(32, 512), torch.rand(32, 512),
torch.rand(32, 512)}

    embeddingModel = nn.Sequential()
    embeddingModel:add(nn.Linear(512, 96)):add(nn.ReLU())

    tripleModel = nn.ParallelTable()
    tripleModel:add(embeddingModel)
    tripleModel:add(embeddingModel:clone('weight', 'bias',
                                         'gradWeight', 'gradBias'))
    tripleModel:add(embeddingModel:clone('weight', 'bias',
                                         'gradWeight', 'gradBias'))

    -- Similar sample distance w.r.t anchor sample
    posDistModel = nn.Sequential()
    posDistModel:add(nn.NarrowTable(1,2)):add(nn.PairwiseDistance())

    -- Different sample distance w.r.t anchor sample
    negDistModel = nn.Sequential()
    negDistModel:add(nn.NarrowTable(2,2)):add(nn.PairwiseDistance())

    distanceModel =
nn.ConcatTable():add(posDistModel):add(negDistModel)
```

```
-- Complete Model
model = nn.Sequential():add(tripleModel):add(distanceModel)

-- DistanceRatioCriterion
criterion = nn.DistanceRatioCriterion(true)

-- Forward & Backward
output = model:forward(sample)
loss   = criterion:forward(output)
dLoss  = criterion:backward(output)
model:backward(sample, dLoss)
```

MarginRankingCriterion

```
criterion = nn.MarginRankingCriterion(margin)
```

Creates a criterion that measures the loss given an input `x = {x1, x2}`, a table of two `Tensor`s of size 1 (they contain only scalars), and a label `y` (`1` or `-1`).
In batch mode, `x` is a table of two `Tensor`s of size `batchsize`, and `y` is a `Tensor` of size `batchsize` containing `1` or `-1` for each corresponding pair of elements in the input `Tensor`.

If `y == 1` then it assumed the first input should be ranked higher (have a larger value) than the second input, and vice-versa for `y == -1`.

The loss function is:

```
loss(x, y) = max(0, -y * (x[1] - x[2]) + margin)
```

For batched inputs, if the internal variable `sizeAverage` is equal to `true`, the loss function averages the loss over the batch samples; if `sizeAverage` is `false`, then the loss function sums over the batch samples. By default, `sizeAverage` equals to `true`.
By default, the losses are averaged over observations for each minibatch. However, if the field `sizeAverage` is set to `false`, the losses are instead summed.

Example

```
p1_mlp = nn.Linear(5, 2)
p2_mlp = p1_mlp:clone('weight', 'bias')

prl = nn.ParallelTable()
prl:add(p1_mlp)
prl:add(p2_mlp)

mlp1 = nn.Sequential()
mlp1:add(prl)
mlp1:add(nn.DotProduct())

mlp2 = mlp1:clone('weight', 'bias')

mlpa = nn.Sequential()
prla = nn.ParallelTable()
prla:add(mlp1)
prla:add(mlp2)
mlpa:add(prla)

crit = nn.MarginRankingCriterion(0.1)

x=torch.randn(5)
y=torch.randn(5)
z=torch.randn(5)

-- Use a typical generic gradient update function
function gradUpdate(mlp, x, y, criterion, learningRate)
   local pred = mlp:forward(x)
   local err = criterion:forward(pred, y)
   local gradCriterion = criterion:backward(pred, y)
   mlp:zeroGradParameters()
   mlp:backward(x, gradCriterion)
   mlp:updateParameters(learningRate)
end

for i = 1, 100 do
   gradUpdate(mlpa, {{x, y}, {x, z}}, 1, crit, 0.01)
   if true then
      o1 = mlp1:forward{x, y}[1]
      o2 = mlp2:forward{x, z}[1]
      o = crit:forward(mlpa:forward{{x, y}, {x, z}}, 1)
      print(o1, o2, o)
   end
end
```

```
print "--"

for i = 1, 100 do
   gradUpdate(mlpa, {{x, y}, {x, z}}, -1, crit, 0.01)
   if true then
      o1 = mlp1:forward{x, y}[1]
      o2 = mlp2:forward{x, z}[1]
      o = crit:forward(mlpa:forward{{x, y}, {x, z}}, -1)
      print(o1, o2, o)
   end
end
```

Transfer Function Layers

Transfer functions are normally used to introduce a non-linearity after a parameterized layer like `Linear` and `SpatialConvolution`.
Non-linearities allows for dividing the problem space into more complex regions than what a simple logistic regressor would permit.

HardTanh

```
f = nn.HardTanh([min_value, max_value[, inplace]])
```

Applies the `HardTanh` function element-wise to the input `Tensor`, thus outputting a `Tensor` of the same dimension.

`HardTanh` is defined as:

```
        ⎧  1, if x >  1
f(x) = ⎨ -1, if x < -1
        ⎩  x, otherwise
```

The range of the linear region `[-1 1]` can be adjusted by specifying arguments in declaration, for example `nn.HardTanh(min_value, max_value)`.
Otherwise, `[min_value max_value]` is set to `[-1 1]` by default.
In-place operation defined by third argument boolean.

```
ii = torch.linspace(-2, 2)
m = nn.HardTanh()
oo = m:forward(ii)
go = torch.ones(100)
gi = m:backward(ii, go)
gnuplot.plot({'f(x)', ii, oo, '+-'}, {'df/dx', ii, gi, '+-'})
gnuplot.grid(true)
```

HardShrink

```
f = nn.HardShrink([lambda])
```

Applies the hard shrinkage function element-wise to the input `Tensor`.
`lambda` is set to `0.5` by default.

`HardShrinkage` operator is defined as:

```
       ⎧ x, if x >  lambda
f(x) = ⎨ x, if x < -lambda
       ⎩ 0, otherwise
```

```
ii = torch.linspace(-2, 2)
m = nn.HardShrink(0.85)
oo = m:forward(ii)
go = torch.ones(100)
gi = m:backward(ii, go)
gnuplot.plot({'f(x)', ii, oo, '+-'}, {'df/dx', ii, gi, '+-'})
gnuplot.grid(true)
```

SoftShrink

```
f = nn.SoftShrink([lambda])
```

Applies the soft shrinkage function element-wise to the input `Tensor`.
`lambda` is set to `0.5` by default.

`SoftShrinkage` operator is defined as:

```
       ⎧ x - lambda, if x >  lambda
f(x) = ⎨ x + lambda, if x < -lambda
```

```
          ⌊ 0, otherwise
```

```
ii = torch.linspace(-2, 2)
m = nn.SoftShrink(0.85)
oo = m:forward(ii)
go = torch.ones(100)
gi = m:backward(ii, go)
gnuplot.plot({'f(x)', ii, oo, '+-'}, {'df/dx', ii, gi, '+-'})
gnuplot.grid(true)
```


SoftMax

```
f = nn.SoftMax()
```

Applies the `SoftMax` function to an n-dimensional input `Tensor`, rescaling them so that the elements of the n-dimensional output Tensor
lie in the range `(0, 1)` and sum to `1`.

`Softmax` is defined as:

```
f_i(x) = exp(x_i - shift) / sum_j exp(x_j - shift)
```

where `shift = max_i(x_i)`.

```
ii = torch.exp(torch.abs(torch.randn(10)))
m = nn.SoftMax()
oo = m:forward(ii)
gnuplot.plot({'Input', ii, '+-'}, {'Output', oo, '+-'})
gnuplot.grid(true)
```


Note that this module doesn't work directly with `ClassNLLCriterion`, which expects the `nn.Log` to be computed between the `SoftMax` and itself.

Use `LogSoftMax` instead (it's faster).

SoftMin

```
f = nn.SoftMin()
```

Applies the `SoftMin` function to an n-dimensional input `Tensor`, rescaling them so that the elements of the n-dimensional output `Tensor` lie in the range `(0,1)` and sum to `1`.

`Softmin` is defined as:

```
f_i(x) = exp(-x_i - shift) / sum_j exp(-x_j - shift)
```

where `shift = max_i(-x_i)`.

```
ii = torch.exp(torch.abs(torch.randn(10)))
m = nn.SoftMin()
oo = m:forward(ii)
gnuplot.plot({'Input', ii, '+-'}, {'Output', oo, '+-'})
gnuplot.grid(true)
```

SoftPlus

```
f = nn.SoftPlus()
```

Applies the `SoftPlus` function to an n-dimensioanl input `Tensor`.
`SoftPlus` is a smooth approximation to the `ReLU` function and can be used to constrain the output of a machine to always be positive.
For numerical stability the implementation reverts to the linear function for inputs above a certain value (20 by default).

`SoftPlus` is defined as:

```
f_i(x) = 1/beta * log(1 + exp(beta * x_i))
```

```
ii = torch.linspace(-3, 3)
m = nn.SoftPlus()
oo = m:forward(ii)
go = torch.ones(100)
gi = m:backward(ii, go)
gnuplot.plot({'f(x)', ii, oo, '+-'}, {'df/dx', ii, gi, '+-'})
gnuplot.grid(true)
```


SoftSign

```
f = nn.SoftSign()
```

Applies the `SoftSign` function to an n-dimensioanl input `Tensor`.

`SoftSign` is defined as:

```
f_i(x) = x_i / (1+|x_i|)
```

```
ii = torch.linspace(-5, 5)
m = nn.SoftSign()
oo = m:forward(ii)
go = torch.ones(100)
gi = m:backward(ii, go)
gnuplot.plot({'f (x)', ii, oo, '+-'}, {'df/dx', ii, gi, '+-'})
gnuplot.grid(true)
```


LogSigmoid

```
f = nn.LogSigmoid()
```

Applies the `LogSigmoid` function to an n-dimensional input `Tensor`.

`LogSigmoid` is defined as:

```
f_i(x) = log(1 / (1 + exp(-x_i)))
```

```
ii = torch.randn(10)
m = nn.LogSigmoid()
oo = m:forward(ii)
go = torch.ones(10)
gi = m:backward(ii, go)
gnuplot.plot({'Input', ii, '+-'}, {'Output', oo, '+-'},
{'gradInput', gi, '+-'})
gnuplot.grid(true)
```

LogSoftMax

```
f = nn.LogSoftMax()
```

Applies the `LogSoftMax` function to an n-dimensional input `Tensor`.

`LogSoftmax` is defined as:

```
f_i(x) = log(1 / a exp(x_i))
```

where `a = sum_j[exp(x_j)]`.

```
ii = torch.randn(10)
m = nn.LogSoftMax()
oo = m:forward(ii)
go = torch.ones(10)
```

```
gi = m:backward(ii, go)
gnuplot.plot({'Input', ii, '+-'}, {'Output', oo, '+-'},
{'gradInput', gi, '+-'})
gnuplot.grid(true)
```

Sigmoid

```
f = nn.Sigmoid()
```

Applies the `Sigmoid` function element-wise to the input `Tensor`, thus outputting a Tensor of the same dimension.

`Sigmoid` is defined as:

```
f(x) = 1 / (1 + exp(-x))
```

```
ii = torch.linspace(-5, 5)
m = nn.Sigmoid()
oo = m:forward(ii)
go = torch.ones(100)
gi = m:backward(ii, go)
gnuplot.plot({'f(x)', ii, oo, '+-'}, {'df/dx', ii, gi, '+-'})
gnuplot.grid(true)
```

Tanh

```
f = nn.Tanh()
```

Applies the `Tanh` function element-wise to the input `Tensor`, thus outputting a `Tensor` of the same dimension.

`Tanh` is defined as:

```
f(x) = (exp(x) - exp(-x)) / (exp(x) + exp(-x))
```

```
ii = torch.linspace(-3, 3)
m = nn.Tanh()
oo = m:forward(ii)
go = torch.ones(100)
gi = m:backward(ii, go)
gnuplot.plot({'f(x)', ii, oo, '+-'}, {'df/dx', ii, gi, '+-'})
gnuplot.grid(true)
```

ReLU

```
f = nn.ReLU([inplace])
```

Applies the rectified linear unit (`ReLU`) function element-wise to the input `Tensor`, thus outputting a `Tensor` of the same dimension.

`ReLU` is defined as:

```
f(x) = max(0, x)
```

Can optionally do its operation in-place without using extra state memory:

```
f = nn.ReLU(true) -- true = in-place, false = keeping separate
state.
```

```
ii = torch.linspace(-3, 3)
m = nn.ReLU()
```

```
oo = m:forward(ii)
go = torch.ones(100)
gi = m:backward(ii, go)
gnuplot.plot({'f(x)', ii, oo, '+-'}, {'df/dx', ii, gi, '+-'})
gnuplot.grid(true)
```

ReLU6

```
f = nn.ReLU6([inplace])
```

Same as `ReLU` except that the rectifying function `f(x)` saturates at `x = 6`.
This layer is useful for training networks that do not loose precision (due to FP saturation) when implemented as FP16.

`ReLU6` is defined as:

```
f(x) = min(max(0, x), 6)
```

Can optionally do its operation in-place without using extra state memory:

```
f = nn.ReLU6(true) -- true = in-place, false = keeping separate
state.
```

```
ii = torch.linspace(-3, 9)
m = nn.ReLU6()
oo = m:forward(ii)
go = torch.ones(100)
gi = m:backward(ii, go)
gnuplot.plot({'f(x)', ii, oo, '+-'}, {'df/dx', ii, gi, '+-'})
gnuplot.grid(true)
```

PReLU

```
f = nn.PReLU()
```

Applies parametric `ReLU`, which parameter varies the slope of the negative part:

`PReLU` is defined as:

```
f(x) = max(0, x) + a * min(0, x)
```

When called without a number on input as `nn.PReLU()` uses shared version, meaning has only one parameter.
Otherwise if called `nn.PReLU(nOutputPlane)` has `nOutputPlane` parameters, one for each input map.
The output dimension is always equal to input dimension.
Note that weight decay should not be used on it.
For reference see Delving Deep into Rectifiers.

RReLU

```
f = nn.RReLU([l, u[, inplace]])
```

Applies the randomized leaky rectified linear unit (`RReLU`) element-wise to the input `Tensor`, thus outputting a `Tensor` of the same dimension.
Informally the `RReLU` is also known as 'insanity' layer.

`RReLU` is defined as:

```
f(x) = max(0,x) + a * min(0, x)
```

where `a ~ U(l, u)`.

In training mode negative inputs are multiplied by a factor `a` drawn from a uniform random distribution `U(l, u)`.
In evaluation mode a `RReLU` behaves like a `LeakyReLU` with a constant mean factor `a = (l + u) / 2`.
By default, `l = 1/8` and `u = 1/3`.
If `l == u` a `RReLU` effectively becomes a `LeakyReLU`.
Regardless of operating in in-place mode a `RReLU` will internally allocate an input-sized `noise` tensor to store random factors for negative inputs.
The `backward()` operation assumes that `forward()` has been called before.

For reference see Empirical Evaluation of Rectified Activations in Convolutional Network.

```
ii = torch.linspace(-3, 3)
m = nn.RReLU()
oo = m:forward(ii):clone()
gi = m:backward(ii, torch.ones(100))
gnuplot.plot({'f(x)', ii, oo, '+-'}, {'df/dx', ii, gi, '+-'})
gnuplot.grid(true)
```


CReLU

```
f = nn.CReLU(nInputDims, [inplace])
```

Applies the Concatenated Rectified Linear Unit (`CReLU`) function to the input Tensor, outputting a `Tensor` with twice as many channels. The parameter `nInputDim` is the number of non-batched dimensions, larger than that value will be considered batches.
`CReLU` is defined as:

```
f(x) = concat(max(0, x), max(0, -x))
```

i.e. `CReLU` applies `ReLU` to the input, `x`, and the negated input, `-x`, and concatenates the output along the 1st non-batched dimension.

```
crelu = nn.CReLU(3)
input = torch.Tensor(2, 3, 20, 20):uniform(-1, 1)
```

```
output = crelu:forward(input)
output:size()
2
6
20
20
[torch.LongStorage of size 4]

input = torch.Tensor(3, 20, 20):uniform(-1, 1)
output = crelu:forward(input)
output:size()
6
20
20
[torch.LongStorage of size 3]
```

For reference see Understanding and Improving Convolutional Neural Networks via Concatenated Rectified Linear Units.

ELU

```
f = nn.ELU([alpha[, inplace]])
```

Applies exponential linear unit (`ELU`), which parameter a varies the convergence value of the exponential function below zero:

`ELU` is defined as:

```
f(x) = max(0, x) + min(0, alpha * (exp(x) - 1))
```

The output dimension is always equal to input dimension.

For reference see Fast and Accurate Deep Network Learning by Exponential Linear Units (ELUs).

```
xs = torch.linspace(-3, 3, 200)
go = torch.ones(xs:size(1))
function f(a) return nn.ELU(a):forward(xs) end
function df(a) local m = nn.ELU(a) m:forward(xs) return
```

```
  m:backward(xs, go) end

  gnuplot.plot({'fw ELU, alpha = 0.1', xs,  f(0.1), '-'},
               {'fw ELU, alpha = 1.0', xs,  f(1.0), '-'},
               {'bw ELU, alpha = 0.1', xs, df(0.1), '-'},
               {'bw ELU, alpha = 1.0', xs, df(1.0), '-'})
  gnuplot.grid(true)
```

LeakyReLU

```
  f = nn.LeakyReLU([negval[, inplace]])
```

Applies `LeakyReLU`, which parameter `negval` sets the slope of the negative part:

`LeakyReLU` is defined as:

```
  f(x) = max(0, x) + negval * min(0, x)
```

Can optionally do its operation in-place without using extra state memory:

```
  f = nn.LeakyReLU(negval, true) -- true = in-place, false = keeping
  separate state.
```

GatedLinearUnit

Applies a Gated Linear unit activation function, which halves the input dimension as follows:

`GatedLinearUnit` is defined as `f([x1, x2])` = `x1 * sigmoid(x2)`

where x1 is the first half of the input vector and x2 is the second half. The multiplication is component-wise, and the input vector must have an even number of elements.

The GatedLinearUnit optionally takes a `dim` parameter, which is the dimension of the input tensor to operate over. It defaults to the last dimension.

SpatialSoftMax

```
f = nn.SpatialSoftMax()
```

Applies `SoftMax` over features to each spatial location (height x width of planes).
The module accepts 1D (vector), 2D (batch of vectors), 3D (vectors in space) or 4D (batch of vectors in space) `Tensor` as input.
Functionally it is equivalent to `SoftMax` when 1D or 2D input is used.
The output dimension is always the same as input dimension.

```
ii = torch.randn(4, 8, 16, 16)  -- batchSize x features x height x
width
m = nn.SpatialSoftMax()
oo = m:forward(ii)
```

SpatialLogSoftMax

Applies LogSoftMax over features to each spatial location (height x width of planes).
The module accepts 1D (vector), 2D (batch of vectors), 3D (vectors in space) or 4D (batch of vectors in space) tensor as input.
Functionally it is equivalent to LogSoftMax when 1D or 2D input is used.
The output dimension is always the same as input dimension.

```
ii=torch.randn(4,8,16,16)  -- batchSize x features x height x width
m=nn.SpatialLogSoftMax()
oo = m:forward(ii)
```

AddConstant

```
f = nn.AddConstant(k[, inplace])
```

Adds a (non-learnable) scalar constant.
This module is sometimes useful for debugging purposes.
Its transfer function is:

```
f(x) = x + k
```

where `k` is a scalar.

Can optionally do its operation in-place without using extra state memory:

```
f = nn.AddConstant(k, true) -- true = in-place, false = keeping
separate state.
```

In-place mode restores the original input value after the backward pass, allowing its use after other in-place modules, like `MulConstant`.

MulConstant

```
f = nn.MulConstant(k[, inplace])
```

Multiplies input `Tensor` by a (non-learnable) scalar constant.
This module is sometimes useful for debugging purposes.
Its transfer function is:

```
f(x) = k * x
```

where `k` is a scalar.

Can optionally do its operation in-place without using extra state memory:

```
m = nn.MulConstant(k, true) -- true = in-place, false = keeping
separate state.
```

In-place mode restores the original input value after the backward pass, allowing its use after other in-place modules, like `AddConstant`.

Convolutional layers

A convolution is an integral that expresses the amount of overlap of one function `g` as it is shifted over another function `f`. It therefore "blends" one function with another. The neural network package supports convolution, pooling, subsampling and other relevant facilities. These are divided based on the dimensionality of the input and output Tensors:

- Temporal Modules apply to sequences with a one-dimensional relationship (e.g. sequences of words, phonemes and letters. Strings of some kind).
  - TemporalConvolution : a 1D convolution over an input sequence ;
  - TemporalSubSampling : a 1D sub-sampling over an input sequence ;
  - TemporalMaxPooling : a 1D max-pooling operation over an input sequence ;
  - LookupTable : a convolution of width `1`, commonly used for word embeddings ;

- Spatial Modules apply to inputs with two-dimensional relationships (e.g. images):
  - SpatialConvolution : a 2D convolution over an input image ;
  - SpatialFullConvolution : a 2D full convolution over an input image ;
  - SpatialDilatedConvolution : a 2D dilated convolution over an input image ;
  - SpatialConvolutionLocal : a 2D locally-connected layer over an input image ;
  - SpatialSubSampling : a 2D sub-sampling over an input image ;
  - SpatialMaxPooling : a 2D max-pooling operation over an input image ;
  - SpatialDilatedMaxPooling : a 2D dilated max-pooling operation over an input image ;
  - SpatialFractionalMaxPooling : a 2D fractional max-pooling operation over an input image ;
  - SpatialAveragePooling : a 2D average-pooling operation over an input image ;
  - SpatialAdaptiveMaxPooling : a 2D max-pooling operation which adapts its parameters dynamically such that the output is of fixed size ;
  - SpatialMaxUnpooling : a 2D max-unpooling operation ;
  - SpatialLPPooling : computes the `p` norm in a convolutional manner on a set of input images ;
  - SpatialConvolutionMap : a 2D convolution that uses a generic connection table ;
  - SpatialZeroPadding : pads a feature map with specified number of zeros ;
  - SpatialReflectionPadding : pads a feature map with the reflection of the input ;
  - SpatialReplicationPadding : pads a feature map with the value at the edge of the input borders ;
  - SpatialSubtractiveNormalization : a spatial subtraction operation on a series of 2D inputs using
  - SpatialCrossMapLRN : a spatial local response normalization between feature maps ;
  - SpatialBatchNormalization: mean/std normalization over the mini-batch inputs

and pixels, with an optional affine transform that follows
a kernel for computing the weighted average in a neighborhood ;
    - SpatialUpsamplingNearest: A simple nearest neighbor upsampler applied to every channel of the feature map.
    - SpatialUpsamplingBilinear: A simple bilinear upsampler applied to every channel of the feature map.

- Volumetric Modules apply to inputs with three-dimensional relationships (e.g. videos) :
    - VolumetricConvolution : a 3D convolution over an input video (a sequence of images) ;
    - VolumetricFullConvolution : a 3D full convolution over an input video (a sequence of images) ;
    - VolumetricDilatedConvolution : a 3D dilated convolution over an input image ;
    - VolumetricMaxPooling : a 3D max-pooling operation over an input video.
    - VolumetricDilatedMaxPooling : a 3D dilated max-pooling operation over an input video ;
    - VolumetricAveragePooling : a 3D average-pooling operation over an input video.
    - VolumetricMaxUnpooling : a 3D max-unpooling operation.
    - VolumetricReplicationPadding : Pads a volumetric feature map with the value at the edge of the input borders. ;

Temporal Modules

Excluding an optional first batch dimension, temporal layers expect a 2D Tensor as input. The first dimension is the number of frames in the sequence (e.g. `nInputFrame`), the last dimension
is the number of features per frame (e.g. `inputFrameSize`). The output will normally have the same number
of dimensions, although the size of each dimension may change. These are commonly used for processing acoustic signals or sequences of words, i.e. in Natural Language Processing.

Note: The LookupTable is special in that while it does output a temporal Tensor of size `nOutputFrame x outputFrameSize`,
its input is a 1D Tensor of indices of size `nIndices`. Again, this is excluding the option first batch dimension.

TemporalConvolution

```
module = nn.TemporalConvolution(inputFrameSize, outputFrameSize,
kW, [dW])
```

Applies a 1D convolution over an input sequence composed of `nInputFrame` frames. The `input` tensor in
`forward(input)` is expected to be a 2D tensor (`nInputFrame x inputFrameSize`) or a 3D tensor (`nBatchFrame x nInputFrame x inputFrameSize`).

The parameters are the following:
* `inputFrameSize`: The input frame size expected in sequences given into `forward()`.
* `outputFrameSize`: The output frame size the convolution layer will produce.
* `kW`: The kernel width of the convolution
* `dW`: The step of the convolution. Default is `1`.

Note that depending of the size of your kernel, several (of the last)
frames of the sequence might be lost. It is up to the user to add proper padding frames in the input
sequences.

If the input sequence is a 2D tensor of dimension `nInputFrame x inputFrameSize`, the output sequence will be
`nOutputFrame x outputFrameSize` where

```
nOutputFrame = (nInputFrame - kW) / dW + 1
```

If the input sequence is a 3D tensor of dimension `nBatchFrame x nInputFrame x inputFrameSize`, the output sequence will be
`nBatchFrame x nOutputFrame x outputFrameSize`.

The parameters of the convolution can be found in `self.weight` (Tensor of
size `outputFrameSize x (kW x inputFrameSize)`) and `self.bias` (Tensor of
size `outputFrameSize`). The corresponding gradients can be found in
`self.gradWeight` and `self.gradBias`.

For a 2D input, the output value of the layer can be precisely described as:

```
output[t][i] = bias[i]
  + sum_j sum_{k=1}^kW weight[i][k][j]
                                * input[dW*(t-1)+k)][j]
```

Here is a simple example:

```
inp=5;  -- dimensionality of one sequence element
outp=1; -- number of derived features for one sequence element
kw=1;   -- kernel only operates on one sequence element per step
dw=1;   -- we step once and go on to the next sequence element

mlp=nn.TemporalConvolution(inp,outp,kw,dw)

x=torch.rand(7,inp) -- a sequence of 7 elements
print(mlp:forward(x))
```

which gives:

```
-0.9109
-0.9872
-0.6808
-0.9403
-0.9680
-0.6901
-0.6387
[torch.Tensor of dimension 7x1]
```

This is equivalent to:

```
weights=torch.reshape(mlp.weight,inp) -- weights applied to all
bias= mlp.bias[1];
for i=1,x:size(1) do -- for each sequence element
   element= x[i]; -- features of ith sequence element
   print(element:dot(weights) + bias)
end
```

which gives:

```
-0.91094998687717
-0.98721705771773
-0.68075004276185
-0.94030132495887
-0.96798754116609
-0.69008470895581
-0.63871422284166
```

TemporalMaxPooling

```
module = nn.TemporalMaxPooling(kW, [dW])
```

Applies 1D max-pooling operation in `kW` regions by step size
`dW` steps. Input sequence composed of `nInputFrame` frames. The `input` tensor in
`forward(input)` is expected to be a 2D tensor (`nInputFrame x inputFrameSize`)
or a 3D tensor (`nBatchFrame x nInputFrame x inputFrameSize`).

If the input sequence is a 2D tensor of dimension `nInputFrame x inputFrameSize` , the
output sequence will be
`nOutputFrame x inputFrameSize` where

```
nOutputFrame = (nInputFrame - kW) / dW + 1
```

TemporalSubSampling

```
module = nn.TemporalSubSampling(inputFrameSize, kW, [dW])
```

Applies a 1D sub-sampling over an input sequence composed of `nInputFrame` frames. The
`input` tensor in
`forward(input)` is expected to be a 2D tensor (`nInputFrame x inputFrameSize`). The
output frame size
will be the same as the input one (`inputFrameSize`).

The parameters are the following:
* `inputFrameSize` : The input frame size expected in sequences given into `forward()` .
* `kW` : The kernel width of the sub-sampling
* `dW` : The step of the sub-sampling. Default is `1` .

Note that depending of the size of your kernel, several (of the last)
frames of the sequence might be lost. It is up to the user to add proper padding frames in the
input
sequences.

If the input sequence is a 2D tensor `nInputFrame x inputFrameSize`, the output sequence will be
`inputFrameSize x nOutputFrame` where

```
nOutputFrame = (nInputFrame - kW) / dW + 1
```

The parameters of the sub-sampling can be found in `self.weight` (Tensor of size `inputFrameSize`) and `self.bias` (Tensor of size `inputFrameSize`). The corresponding gradients can be found in `self.gradWeight` and `self.gradBias`.

The output value of the layer can be precisely described as:

```
output[t][i] = bias[i] + weight[i] * sum_{k=1}^kW input[dW*(t-1)+k]
[i]
```

LookupTable

```
module = nn.LookupTable(nIndex, size, [paddingValue], [maxNorm],
[normType])
```

This layer is a particular case of a convolution, where the width of the convolution would be `1`. When calling `forward(input)`, it assumes `input` is a 1D or 2D tensor filled with indices. If the input is a matrix, then each row is assumed to be an input sample of given batch. Indices start
at `1` and can go up to `nIndex`. For each index, it outputs a corresponding `Tensor` of size specified by `size`.

LookupTable can be very slow if a certain input occurs frequently compared to other inputs; this is often the case for input padding. During the backward step, there is a separate thread for each input symbol which results in a bottleneck for frequent inputs.
generating a `n x size1 x size2 x ... x sizeN` tensor, where `n`
is the size of a 1D `input` tensor.

Again with a 1D input, when only `size1` is provided, the `forward(input)` is equivalent to performing the following matrix-matrix multiplication in an efficient manner:

```
P M
```

where `M` is a 2D matrix of size `nIndex x size1` containing the parameters of the lookup-table and
`P` is a 2D matrix of size `n x nIndex`, where for each `i` th row vector, every element is zero except the one at index `input[i]` where it is `1`.

1D example:

```
-- a lookup table containing 10 tensors of size 3
module = nn.LookupTable(10, 3)

input = torch.Tensor{1,2,1,10}
print(module:forward(input))
```

Outputs something like:

```
-1.4415 -0.1001 -0.1708
-0.6945 -0.4350  0.7977
-1.4415 -0.1001 -0.1708
-0.0745  1.9275  1.0915
[torch.DoubleTensor of dimension 4x3]
```

Note that the first row vector is the same as the 3rd one!

Given a 2D input tensor of size `m x n`, the output is a `m x n x size` tensor, where `m` is the number of samples in
the batch and `n` is the number of indices per sample.

2D example:

```
-- a lookup table containing 10 tensors of size 3
module = nn.LookupTable(10, 3)

-- a batch of 2 samples of 4 indices each
input = torch.Tensor({{1,2,4,5},{4,3,2,10}})
print(module:forward(input))
```

Outputs something like:

```
(1,.,.) =
 -0.0570 -1.5354  1.8555
 -0.9067  1.3392  0.6275
  1.9662  0.4645 -0.8111
  0.1103  1.7811  1.5969

(2,.,.) =
  1.9662  0.4645 -0.8111
  0.0026 -1.4547 -0.5154
 -0.9067  1.3392  0.6275
 -0.0193 -0.8641  0.7396
[torch.DoubleTensor of dimension 2x4x3]
```

LookupTable supports max-norm regularization. One can activate the max-norm constraints by setting non-nil maxNorm in constructor or using setMaxNorm function. In the implementation,
the max-norm constraint is enforced in the forward pass. That is the output of the LookupTable always obeys the max-norm constraint, even though the module weights may temporarily exceed
the max-norm constraint.

max-norm regularization example:

```
-- a lookup table with max-norm constraint: 2-norm <= 1
module = nn.LookupTable(10, 3, 0, 1, 2)
input = torch.Tensor{1,2,1,10}
print(module.weight)
-- output of the module always obey max-norm constraint
print(module:forward(input))
-- the rows accessed should be re-normalized
print(module.weight)
```

Outputs something like:

```
 0.2194  1.4759 -1.1829
 0.7069  0.2436  0.9876
-0.2955  0.3267  1.1844
-0.0575 -0.2957  1.5079
-0.2541  0.5331 -0.0083
 0.8005 -1.5994 -0.4732
-0.0065  2.3441 -0.6354
 0.2910  0.4230  0.0975
```

```
 1.2662  1.1846  1.0114
-0.4095 -1.0676 -0.9056
[torch.DoubleTensor of size 10x3]

 0.1152  0.7751 -0.6212
 0.5707  0.1967  0.7973
 0.1152  0.7751 -0.6212
-0.2808 -0.7319 -0.6209
[torch.DoubleTensor of size 4x3]

 0.1152  0.7751 -0.6212
 0.5707  0.1967  0.7973
-0.2955  0.3267  1.1844
-0.0575 -0.2957  1.5079
-0.2541  0.5331 -0.0083
 0.8005 -1.5994 -0.4732
-0.0065  2.3441 -0.6354
 0.2910  0.4230  0.0975
 1.2662  1.1846  1.0114
-0.2808 -0.7319 -0.6209
[torch.DoubleTensor of size 10x3]
```

Note that the 1st, 2nd and 10th rows of the module.weight are updated to obey the max-norm constraint, since their indices appear in the “input”.

Spatial Modules

Excluding an optional batch dimension, spatial layers expect a 3D Tensor as input. The first dimension is the number of features (e.g. `frameSize`), the last two dimensions are spatial (e.g. `height x width`). These are commonly used for processing images.

SpatialConvolution

```
module = nn.SpatialConvolution(nInputPlane, nOutputPlane, kW, kH,
[dW], [dH], [padW], [padH])
```

Applies a 2D convolution over an input image composed of several input planes. The `input`

tensor in
`forward(input)` is expected to be a 3D tensor (`nInputPlane x height x width`).

The parameters are the following:
* `nInputPlane` : The number of expected input planes in the image given into `forward()`.
* `nOutputPlane` : The number of output planes the convolution layer will produce.
* `kW` : The kernel width of the convolution
* `kH` : The kernel height of the convolution
* `dW` : The step of the convolution in the width dimension. Default is `1`.
* `dH` : The step of the convolution in the height dimension. Default is `1`.
* `padW` : Additional zeros added to the input plane data on both sides of width axis. Default is `0`. `(kW-1)/2` is often used here.
* `padH` : Additional zeros added to the input plane data on both sides of height axis. Default is `0`. `(kH-1)/2` is often used here.

Note that depending of the size of your kernel, several (of the last)
columns or rows of the input image might be lost. It is up to the user to
add proper padding in images.

If the input image is a 3D tensor `nInputPlane x height x width`, the output image size will be `nOutputPlane x oheight x owidth` where

```
owidth  = floor((width  + 2*padW - kW) / dW + 1)
oheight = floor((height + 2*padH - kH) / dH + 1)
```

The parameters of the convolution can be found in `self.weight` (Tensor of size `nOutputPlane x nInputPlane x kH x kW`) and `self.bias` (Tensor of size `nOutputPlane`). The corresponding gradients can be found in `self.gradWeight` and `self.gradBias`.

The output value of the layer can be precisely described as:

```
output[i][j][k] = bias[k]
  + sum_l sum_{s=1}^kW sum_{t=1}^kH weight[s][t][l][k]
                                    * input[dW*(i-1)+s)][dH*
(j-1)+t][l]
```

SpatialConvolutionMap

```
module = nn.SpatialConvolutionMap(connectionMatrix, kW, kH, [dW],
[dH])
```

This class is a generalization of nn.SpatialConvolution. It uses a generic connection table between input and output features. The nn.SpatialConvolution is equivalent to using a full connection table. One can specify different types of connection tables.

Full Connection Table

```
table = nn.tables.full(nin,nout)
```

This is a precomputed table that specifies connections between every input and output node.

One to One Connection Table

```
table = nn.tables.oneToOne(n)
```

This is a precomputed table that specifies a single connection to each output node from corresponding input node.

Random Connection Table

```
table = nn.tables.random(nin,nout, nto)
```

This table is randomly populated such that each output unit has `nto` incoming connections. The algorithm tries to assign uniform number of outgoing connections to each input node if possible.

SpatialFullConvolution

```
module = nn.SpatialFullConvolution(nInputPlane, nOutputPlane, kW,
kH, [dW], [dH], [padW], [padH], [adjW], [adjH])
```

Applies a 2D full convolution over an input image composed of several input planes. The `input` tensor in
`forward(input)` is expected to be a 3D or 4D tensor. Note that instead of setting `adjW` and `adjH`, SpatialFullConvolution also accepts a table input with two tensors: `{convInput, sizeTensor}` where `convInput` is the standard input on which the full convolution is applied, and the size of `sizeTensor` is used to set the size of the output. Using the two-input version of forward
will ignore the `adjW` and `adjH` values used to construct the module. The layer can be used without a bias by module:noBias().

Other frameworks call this operation “In-network Upsampling”, “Fractionally-strided convolution”, “Backwards Convolution,” “Deconvolution”, or “Upconvolution.”

The parameters are the following:
* `nInputPlane`: The number of expected input planes in the image given into `forward()`.
* `nOutputPlane`: The number of output planes the convolution layer will produce.
* `kW`: The kernel width of the convolution
* `kH`: The kernel height of the convolution
* `dW`: The step of the convolution in the width dimension. Default is `1`.
* `dH`: The step of the convolution in the height dimension. Default is `1`.
* `padW`: Additional zeros added to the input plane data on both sides of width axis. Default is `0`. `(kW-1)/2` is often used here.
* `padH`: Additional zeros added to the input plane data on both sides of height axis. Default is `0`. `(kH-1)/2` is often used here.
* `adjW`: Extra width to add to the output image. Default is `0`. Cannot be greater than dW-1.
* `adjH`: Extra height to add to the output image. Default is `0`. Cannot be greater than dH-1.

If the input image is a 3D tensor `nInputPlane x height x width`, the output image size will be `nOutputPlane x oheight x owidth` where

```
owidth  = (width  - 1) * dW - 2*padW + kW + adjW
oheight = (height - 1) * dH - 2*padH + kH + adjH
```

Further information about the full convolution can be found in the following paper: Fully Convolutional Networks for Semantic Segmentation.

SpatialDilatedConvolution

```
module = nn.SpatialDilatedConvolution(nInputPlane, nOutputPlane,
kW, kH, [dW], [dH], [padW], [padH], [dilationW], [dilationH])
```

Also sometimes referred to as **atrous convolution**.
Applies a 2D dilated convolution over an input image composed of several input planes. The `input` tensor in
`forward(input)` is expected to be a 3D or 4D tensor.

The parameters are the following:
* `nInputPlane` : The number of expected input planes in the image given into `forward()` .
* `nOutputPlane` : The number of output planes the convolution layer will produce.
* `kW` : The kernel width of the convolution
* `kH` : The kernel height of the convolution
* `dW` : The step of the convolution in the width dimension. Default is `1` .
* `dH` : The step of the convolution in the height dimension. Default is `1` .
* `padW` : Additional zeros added to the input plane data on both sides of width axis. Default is `0` . `(kW-1)/2` is often used here.
* `padH` : Additional zeros added to the input plane data on both sides of height axis. Default is `0` . `(kH-1)/2` is often used here.
* `dilationW` : The number of pixels to skip. Default is `1` . `1` makes it a SpatialConvolution
* `dilationH` : The number of pixels to skip. Default is `1` . `1` makes it a SpatialConvolution

If the input image is a 3D tensor `nInputPlane x height x width` , the output image size will be `nOutputPlane x oheight x owidth` where

```
owidth  = floor(width + 2 * padW - dilationW * (kW-1) - 1) / dW + 1
oheight = floor(height + 2 * padH - dilationH * (kH-1) - 1) / dH +
1
```

Further information about the dilated convolution can be found in the following paper: Multi-Scale Context Aggregation by Dilated Convolutions.

SpatialConvolutionLocal

```
module = nn.SpatialConvolutionLocal(nInputPlane, nOutputPlane, iW,
iH, kW, kH, [dW], [dH], [padW], [padH])
```

Applies a 2D locally-connected layer over an input image composed of several input planes. The

`input` tensor in
`forward(input)` is expected to be a 3D or 4D tensor.

A locally-connected layer is similar to a convolution layer but without weight-sharing.

The parameters are the following:
* `nInputPlane` : The number of expected input planes in the image given into `forward()` .
* `nOutputPlane` : The number of output planes the locally-connected layer will produce.
* `iW` : The input width.
* `iH` : The input height.
* `kW` : The kernel width.
* `kH` : The kernel height.
* `dW` : The step in the width dimension. Default is `1` .
* `dH` : The step in the height dimension. Default is `1` .
* `padW` : Additional zeros added to the input plane data on both sides of width axis. Default is `0` .
* `padH` : Additional zeros added to the input plane data on both sides of height axis. Default is `0` .

If the input image is a 3D tensor `nInputPlane x iH x iW` , the output image size
will be `nOutputPlane x oH x oW` where

```
oW  = floor((iW  + 2*padW - kW) / dW + 1)
oH = floor((iH + 2*padH - kH) / dH + 1)
```

SpatialLPPooling

```
module = nn.SpatialLPPooling(nInputPlane, pnorm, kW, kH, [dW],
[dH])
```

Computes the `p` norm in a convolutional manner on a set of 2D input planes.

SpatialMaxPooling

```
module = nn.SpatialMaxPooling(kW, kH [, dW, dH, padW, padH])
```

Applies 2D max-pooling operation in `kWxkH` regions by step size `dWxdH` steps. The number of output features is equal to the number of input planes.

If the input image is a 3D tensor `nInputPlane x height x width`, the output image size will be `nOutputPlane x oheight x owidth` where

```
owidth  = op((width  + 2*padW - kW) / dW + 1)
oheight = op((height + 2*padH - kH) / dH + 1)
```

`op` is a rounding operator. By default, it is `floor`. It can be changed by calling `:ceil()` or `:floor()` methods.

SpatialDilatedMaxPooling

```
module = nn.SpatialDilatedMaxPooling(kW, kH [, dW, dH, padW, padH,
dilationW, dilationH])
```

Also sometimes referred to as **atrous pooling**.
Applies 2D dilated max-pooling operation in `kWxkH` regions by step size `dWxdH` steps. The number of output features is equal to the number of input planes. If `dilationW` and `dilationH` are not provided, this is equivalent to performing normal `nn.SpatialMaxPooling`.

If the input image is a 3D tensor `nInputPlane x height x width`, the output image size will be `nOutputPlane x oheight x owidth` where

```
owidth  = op((width - (dilationW * (kW - 1) + 1) + 2*padW) / dW +
1)
oheight = op((height - (dilationH * (kH - 1) + 1) + 2*padH) / dH +
1)
```

`op` is a rounding operator. By default, it is `floor`. It can be changed by calling `:ceil()` or `:floor()` methods.

SpatialFractionalMaxPooling

```
module = nn.SpatialFractionalMaxPooling(kW, kH, outW, outH)
--   the output should be the exact size (outH x outW)
OR
module = nn.SpatialFractionalMaxPooling(kW, kH, ratioW, ratioH)
--   the output should be the size (floor(inH x ratioH) x floor(inW
x ratioW))
--   ratios are numbers between (0, 1) exclusive
```

Applies 2D Fractional max-pooling operation as described in the paper "Fractional Max Pooling" by Ben Graham in the "pseudorandom" mode.

The max-pooling operation is applied in `kWxkH` regions by a stochastic step size determined by the target output size.
The number of output features is equal to the number of input planes.

There are two constructors available.

Constructor 1:

```
module = nn.SpatialFractionalMaxPooling(kW, kH, outW, outH)
```

Constructor 2:

```
module = nn.SpatialFractionalMaxPooling(kW, kH, ratioW, ratioH)
```

If the input image is a 3D tensor `nInputPlane x height x width`, the output image size will be `nOutputPlane x oheight x owidth`

where

```
owidth  = floor(width * ratioW)
oheight = floor(height * ratioH)
```

ratios are numbers between (0, 1) exclusive

SpatialAveragePooling

```
module = nn.SpatialAveragePooling(kW, kH [, dW, dH, padW, padH])
```

Applies 2D average-pooling operation in `kWxkH` regions by step size `dWxdH` steps. The number of output features is equal to the number of input planes.

If the input image is a 3D tensor `nInputPlane x height x width`, the output image size will be `nOutputPlane x oheight x owidth` where

```
owidth  = op((width  + 2*padW - kW) / dW + 1)
oheight = op((height + 2*padH - kH) / dH + 1)
```

`op` is a rounding operator. By default, it is `floor`. It can be changed by calling `:ceil()` or `:floor()` methods.

By default, the output of each pooling region is divided by the number of elements inside the padded image (which is usually `kW*kH`, except in some corner cases in which it can be smaller). You can also divide by the number of elements inside the original non-padded image. To switch between different division factors, call `:setCountIncludePad()` or `:setCountExcludePad()`. If `padW=padH=0`, both options give the same results.

SpatialAdaptiveMaxPooling

```
module = nn.SpatialAdaptiveMaxPooling(W, H)
```

Applies 2D max-pooling operation in an image such that the output is of size `WxH`, for any input size. The number of output features is equal to the number of input planes.

For an output of dimensions `(owidth,oheight)`, the indexes of the pooling region `(j,i)` in the input image of dimensions `(iwidth,iheight)` are given by:

```
x_j_start = floor((j   /owidth)  * iwidth)
x_j_end   = ceil(((j+1)/owidth)  * iwidth)

y_i_start = floor((i   /oheight) * iheight)
y_i_end   = ceil(((i+1)/oheight) * iheight)
```

SpatialMaxUnpooling

```
module = nn.SpatialMaxUnpooling(poolingModule)
```

Applies 2D “max-unpooling” operation using the indices previously computed
by the SpatialMaxPooling module `poolingModule`.

When `B = poolingModule:forward(A)` is called, the indices of the maximal
values (corresponding to their position within each map) are stored:
`B[{n,k,i,j}] = A[{n,k,indices[{n,k,i}],indices[{n,k,j}]}]`.
If `C` is a tensor of same size as `B`, `module:updateOutput(C)` outputs a
tensor `D` of same size as `A` such that:
`D[{n,k,indices[{n,k,i}],indices[{n,k,j}]}] = C[{n,k,i,j}]`.

Module inspired by:
“Visualizing and understanding convolutional networks” (2014)
by Matthew Zeiler, Rob Fergus

SpatialSubSampling

```
module = nn.SpatialSubSampling(nInputPlane, kW, kH, [dW], [dH])
```

Applies a 2D sub-sampling over an input image composed of several input planes. The `input` tensor in
`forward(input)` is expected to be a 3D tensor (`nInputPlane x height x width`). The number of output
planes will be the same as `nInputPlane`.

The parameters are the following:
* `nInputPlane`: The number of expected input planes in the image given into `forward()`.
* `kW`: The kernel width of the sub-sampling
* `kH`: The kernel height of the sub-sampling
* `dW`: The step of the sub-sampling in the width dimension. Default is `1`.
* `dH`: The step of the sub-sampling in the height dimension. Default is `1`.

Note that depending of the size of your kernel, several (of the last)
columns or rows of the input image might be lost. It is up to the user to
add proper padding in images.

If the input image is a 3D tensor `nInputPlane x height x width`, the output image size will be `nInputPlane x oheight x owidth` where

```
owidth  = (width  - kW) / dW + 1
oheight = (height - kH) / dH + 1 .
```

The parameters of the sub-sampling can be found in `self.weight` (Tensor of size `nInputPlane`) and `self.bias` (Tensor of size `nInputPlane`). The corresponding gradients can be found in `self.gradWeight` and `self.gradBias`.

The output value of the layer can be precisely described as:

```
output[i][j][k] = bias[k]
  + weight[k] sum_{s=1}^kW sum_{t=1}^kH input[dW*(i-1)+s)][dH*
(j-1)+t][k]
```

SpatialUpSamplingNearest

```
module = nn.SpatialUpSamplingNearest(scale)
```

Applies a 2D up-sampling over an input image composed of several input planes. The `input` tensor in
`forward(input)` is expected to be a 3D or 4D tensor (i.e. for 4D: `nBatchPlane x nInputPlane x height x width`). The number of output planes will be the same. The v dimension is assumed to be the second last dimension (i.e. for 4D it will be the 3rd dim), and the u dimension is assumed to be the last dimension.

The parameters are the following:
* `scale`: The upscale ratio. Must be a positive integer

The up-scaling method is simple nearest neighbor, ie:

```
output(u,v) = input(floor((u-1)/scale)+1, floor((v-1)/scale)+1)
```

Where `u` and `v` are index from 1 (as per lua convention). There are no learnable parameters.

SpatialUpSamplingBilinear

```
module = nn.SpatialUpSamplingBilinear(scale)
module = nn.SpatialUpSamplingBilinear({oheight=H, owidth=W})
```

Applies a 2D up-sampling over an input image composed of several input planes. The `input` tensor in
`forward(input)` is expected to be a 3D or 4D tensor (i.e. for 4D: `nBatchPlane x nInputPlane x height x width`). The number of output planes will be the same. The v dimension is assumed to be the second last dimension (i.e. for 4D it will be the 3rd dim), and the u dimension is assumed to be the last dimension.

The parameters are the following:
* `scale` : The upscale ratio. Must be a positive integer
* Or a table `{oheight=H, owidth=W}` : The required output height and width, should be positive integers.

The up-scaling method is bilinear.
If `scale` is specified, given an input of height iH and width iW, output height and width will be:

```
oH = (iH - 1)(scale - 1) + iH
oW = (iW - 1)(scale - 1) + iW
```

There are no learnable parameters.

SpatialZeroPadding

```
module = nn.SpatialZeroPadding(padLeft, padRight, padTop,
padBottom)
```

Each feature map of a given input is padded with specified number of
zeros. If padding values are negative, then input is cropped.

SpatialReflectionPadding

```
module = nn.SpatialReflectionPadding(padLeft, padRight, padTop,
padBottom)
```

Each feature map of a given input is padded with the reflection of the input boundary

SpatialReplicationPadding

```
module = nn.SpatialReplicationPadding(padLeft, padRight, padTop,
padBottom)
```

Each feature map of a given input is padded with the replication of the input boundary

SpatialSubtractiveNormalization

```
module = nn.SpatialSubtractiveNormalization(ninputplane, kernel)
```

Applies a spatial subtraction operation on a series of 2D inputs using `kernel` for computing the weighted average in a neighborhood. The neighborhood is defined for a local spatial region that is the size as kernel and across all features. For a an input image, since there is only one feature, the region is only spatial. For an RGB image, the weighted average is taken over RGB channels and a spatial region.

If the `kernel` is 1D, then it will be used for constructing and seperable 2D kernel. The operations will be much more efficient in this case.

The kernel is generally chosen as a gaussian when it is believed that the correlation of two pixel locations decrease with increasing distance. On the feature dimension, a uniform average is used since the weighting across features is not known.

For this example we use an external package
image

```
require 'image'
require 'nn'
```

```
lena = image.rgb2y(image.lena())
ker = torch.ones(11)
m=nn.SpatialSubtractiveNormalization(1,ker)
processed = m:forward(lena)
w1=image.display(lena)
w2=image.display(processed)
```

SpatialCrossMapLRN

```
module = nn.SpatialCrossMapLRN(size [,alpha] [,beta] [,k])
```

Applies Spatial Local Response Normalization between different feature maps.
By default, `alpha = 0.0001`, `beta = 0.75` and `k = 1`

The operation implemented is:

```
                              x_f
y_f =  ------------------------------------------------
        (k+(alpha/size)* sum_{l=l1 to l2} (x_l^2))^beta
```

where `x_f` is the input at spatial locations `h,w` (not shown for simplicity) and feature map

`f`,
`l1` corresponds to `max(0,f-ceil(size/2))` and `l2` to `min(F, f-ceil(size/2) + size)`. Here, `F`
is the number of feature maps.
More information can be found here.

SpatialBatchNormalization

`module` = `nn.SpatialBatchNormalization(N [,eps] [, momentum] [,affine])`
where N = number of input feature maps
eps is a small value added to the standard-deviation to avoid divide-by-zero. Defaults to 1e-5
`affine` is a boolean. When set to false, the learnable affine transform is disabled. Defaults to true

Implements Batch Normalization as described in the paper:
"Batch Normalization: Accelerating Deep Network Training
by Reducing Internal Covariate Shift"
by Sergey Ioffe, Christian Szegedy

The operation implemented is:

```
     y =     ( x - mean(x) )
          -------------------- * gamma + beta
         standard-deviation(x)
```

where the mean and standard-deviation are calculated per feature-map over the mini-batches and pixels
and where gamma and beta are learnable parameter vectors of size N (where N = number of feature maps).
The learning of gamma and beta is optional.

In training time, this layer keeps a running estimate of it's computed mean and std.
The running sum is kept with a default momentup of 0.1 (unless over-ridden)
In test time, this running mean/std is used to normalize.

The module only accepts 4D inputs.

```
-- with learnable parameters
model = nn.SpatialBatchNormalization(m)
```

```
A = torch.randn(b, m, h, w)
C = model:forward(A)  -- C will be of size `b x m x h x w`

-- without learnable parameters
model = nn.SpatialBatchNormalization(m, nil, nil, false)
A = torch.randn(b, m, h, w)
C = model:forward(A)  -- C will be of size `b x m x h x w`
```

Volumetric Modules

Excluding an optional batch dimension, volumetric layers expect a 4D Tensor as input. The first dimension is the number of features (e.g. `frameSize`), the second is sequential (e.g. `time`) and the
last two dimensions are spatial (e.g. `height x width`). These are commonly used for processing videos (sequences of images).

VolumetricConvolution

```
module = nn.VolumetricConvolution(nInputPlane, nOutputPlane, kT,
kW, kH [, dT, dW, dH, padT, padW, padH])
```

Applies a 3D convolution over an input image composed of several input planes. The `input` tensor in
`forward(input)` is expected to be a 4D tensor (`nInputPlane x time x height x width`).

The parameters are the following:
* `nInputPlane`: The number of expected input planes in the image given into `forward()`.
* `nOutputPlane`: The number of output planes the convolution layer will produce.
* `kT`: The kernel size of the convolution in time
* `kW`: The kernel width of the convolution
* `kH`: The kernel height of the convolution
* `dT`: The step of the convolution in the time dimension. Default is `1`.
* `dW`: The step of the convolution in the width dimension. Default is `1`.
* `dH`: The step of the convolution in the height dimension. Default is `1`.
* `padT`: Additional zeros added to the input plane data on both sides of time axis. Default is `0`.

`(kT-1)/2` is often used here.
* `padW` : Additional zeros added to the input plane data on both sides of width axis. Default is `0`. `(kW-1)/2` is often used here.
* `padH` : Additional zeros added to the input plane data on both sides of height axis. Default is `0`. `(kH-1)/2` is often used here.

Note that depending of the size of your kernel, several (of the last)
columns or rows of the input image might be lost. It is up to the user to
add proper padding in images.

If the input image is a 4D tensor `nInputPlane x time x height x width`, the output image size
will be `nOutputPlane x otime x oheight x owidth` where

```
otime  = floor((time  + 2*padT - kT) / dT + 1)
owidth  = floor((width  + 2*padW - kW) / dW + 1)
oheight  = floor((height  + 2*padH - kH) / dH + 1)
```

The parameters of the convolution can be found in `self.weight` (Tensor of size `nOutputPlane x nInputPlane x kT x kH x kW`) and `self.bias` (Tensor of size `nOutputPlane`). The corresponding gradients can be found in `self.gradWeight` and `self.gradBias`.

VolumetricFullConvolution

```
module = nn.VolumetricFullConvolution(nInputPlane, nOutputPlane,
kT, kW, kH, [dT], [dW], [dH], [padT], [padW], [padH], [adjT],
[adjW], [adjH])
```

Applies a 3D full convolution over an input image composed of several input planes. The `input` tensor in
`forward(input)` is expected to be a 4D or 5D tensor. Note that instead of setting `adjT`, `adjW` and `adjH`, VolumetricFullConvolution also accepts a table input with two tensors: `{convInput, sizeTensor}` where `convInput` is the standard input on which the full convolution is applied, and the size of `sizeTensor` is used to set the size of the output. Using the two-input version of forward
will ignore the `adjT`, `adjW` and `adjH` values used to construct the module.

This can be used as 3D deconvolution, or 3D upsampling. So that the 3D FCN can be easly implemented.

The parameters are the following:
* `nInputPlane` : The number of expected input planes in the image given into `forward()` .
* `nOutputPlane` : The number of output planes the convolution layer will produce.
* `kT` : The kernel depth of the convolution
* `kW` : The kernel width of the convolution
* `kH` : The kernel height of the convolution
* `dT` : The step of the convolution in the depth dimension. Default is `1` .
* `dW` : The step of the convolution in the width dimension. Default is `1` .
* `dH` : The step of the convolution in the height dimension. Default is `1` .
* `padT` : Additional zeros added to the input plane data on both sides of time (depth) axis. Default is `0` . `(kT-1)/2` is often used here.
* `padW` : Additional zeros added to the input plane data on both sides of width axis. Default is `0` . `(kW-1)/2` is often used here.
* `padH` : Additional zeros added to the input plane data on both sides of height axis. Default is `0` . `(kH-1)/2` is often used here.
* `adjT` : Extra depth to add to the output image. Default is `0` . Cannot be greater than dT-1.
* `adjW` : Extra width to add to the output image. Default is `0` . Cannot be greater than dW-1.
* `adjH` : Extra height to add to the output image. Default is `0` . Cannot be greater than dH-1.

If the input image is a 3D tensor `nInputPlane x depth x height x width` , the output image size
will be `nOutputPlane x odepth x oheight x owidth` where

```
odepth  = (depth  - 1) * dT - 2*padT + kT + adjT
owidth  = (width  - 1) * dW - 2*padW + kW + adjW
oheight = (height - 1) * dH - 2*padH + kH + adjH
```

VolumetricDilatedConvolution

```
module = nn.VolumetricDilatedConvolution(nInputPlane, nOutputPlane,
kT, kW, kH, [dT], [dW], [dH], [padT], [padW], [padH], [dilationT],
[dilationW], [dilationH])
```

Applies a 3D dilated convolution over an input image composed of several input planes. The `input` tensor in
`forward(input)` is expected to be a 4D or 5D tensor.

The parameters are the following:
* `nInputPlane` : The number of expected input planes in the image given into `forward()` .

* `nOutputPlane` : The number of output planes the convolution layer will produce.
* `kT` : The kernel depth of the convolution
* `kW` : The kernel width of the convolution
* `kH` : The kernel height of the convolution
* `dT` : The step of the convolution in the depth dimension. Default is `1`.
* `dW` : The step of the convolution in the width dimension. Default is `1`.
* `dH` : The step of the convolution in the height dimension. Default is `1`.
* `padT` : Additional zeros added to the input plane data on both sides of time (depth) axis. Default is `0`. `(kT-1)/2` is often used here.
* `padW` : Additional zeros added to the input plane data on both sides of width axis. Default is `0`. `(kW-1)/2` is often used here.
* `padH` : Additional zeros added to the input plane data on both sides of height axis. Default is `0`. `(kH-1)/2` is often used here.
* `dilationT` : The number of pixels to skip. Default is `1`. `1` makes it a VolumetricConvolution
* `dilationW` : The number of pixels to skip. Default is `1`. `1` makes it a VolumetricConvolution
* `dilationH` : The number of pixels to skip. Default is `1`. `1` makes it a VolumetricConvolution

If the input image is a 4D tensor `nInputPlane x depth x height x width`, the output image size
will be `nOutputPlane x odepth x oheight x owidth` where

```
odepth  = floor(depth + 2 * padT - dilationT * (kT-1) + 1) / dT + 1
owidth  = floor(width + 2 * padW - dilationW * (kW-1) + 1) / dW + 1
oheight = floor(height + 2 * padH - dilationH * (kH-1) + 1) / dH +
1
```

Further information about the dilated convolution can be found in the following paper: Multi-Scale Context Aggregation by Dilated Convolutions.

VolumetricMaxPooling

```
module = nn.VolumetricMaxPooling(kT, kW, kH [, dT, dW, dH, padT,
padW, padH])
```

Applies 3D max-pooling operation in `kTxkWxkH` regions by step size `dTxdWxdH` steps. The number of output features is equal to the number of input planes / dT. The input can optionally be padded with zeros. Padding should be smaller than half of kernel size. That is, `padT < kT/2`, `padW < kW/2` and `padH < kH/2`.

VolumetricDilatedMaxPooling

```
module = nn.VolumetricDilatedMaxPooling(kT, kW, kH [, dT, dW, dH,
padT, padW, padH, dilationT, dilationW, dilationH])
```

Also sometimes referred to as **atrous pooling**.
Applies 3D dilated max-pooling operation in `kTxkWxkH` regions by step size `dTxdWxdH` steps. The number of output features is equal to the number of input planes. If `dilationT`, `dilationW` and `dilationH` are not provided, this is equivalent to performing normal `nn.VolumetricMaxPooling`.

If the input image is a 4D tensor `nInputPlane x depth x height x width`, the output image size will be `nOutputPlane x otime x oheight x owidth` where

```
otime  = op((depth - (dilationT * (kT - 1) + 1) + 2*padT) / dT + 1)
owidth  = op((width - (dilationW * (kW - 1) + 1) + 2*padW) / dW +
1)
oheight = op((height - (dilationH * (kH - 1) + 1) + 2*padH) / dH +
1)
```

`op` is a rounding operator. By default, it is `floor`. It can be changed by calling `:ceil()` or `:floor()` methods.

VolumetricAveragePooling

```
module = nn.VolumetricAveragePooling(kT, kW, kH [, dT, dW, dH])
```

Applies 3D average-pooling operation in `kTxkWxkH` regions by step size `dTxdWxdH` steps. The number of output features is equal to the number of input planes / dT.

VolumetricMaxUnpooling

```
module = nn.VolumetricMaxUnpooling(poolingModule)
```

Applies 3D “max-unpooling” operation using the indices previously computed by the VolumetricMaxPooling module `poolingModule`.

When `B = poolingModule:forward(A)` is called, the indices of the maximal values (corresponding to their position within each map) are stored: `B[{n,k,t,i,j}] = A[{n,k,indices[{n,k,t}],indices[{n,k,i}],indices[{n,k,j}]}]`. If `C` is a tensor of same size as `B`, `module:updateOutput(C)` outputs a tensor `D` of same size as `A` such that: `D[{n,k,indices[{n,k,t}],indices[{n,k,i}],indices[{n,k,j}]}] = C[{n,k,t,i,j}]`.

VolumetricReplicationPadding

```
module = nn.VolumetricReplicationPadding(padLeft, padRight, padTop,
padBottom,
                                         padFront, padBack)
```

Each feature map of a given input is padded with the replication of the input boundary.

Training a neural network

Training a neural network is easy with a simple `for` loop. Typically however we would use the `optim` optimizer, which implements some cool functionalities, like Nesterov momentum,
adagrad and
adam.

We will demonstrate using a for-loop first, to show the low-level view of what happens in training. StochasticGradient, a simple class
which does the job for you, is provided as standard. Finally, `optim` is a powerful module, that provides multiple optimization algorithms.

Example of manual training of a neural network

We show an example here on a classical XOR problem.

Neural Network

We create a simple neural network with one hidden layer.

```
require "nn"
mlp = nn.Sequential();  -- make a multi-layer perceptron
inputs = 2; outputs = 1; HUs = 20; -- parameters
mlp:add(nn.Linear(inputs, HUs))
mlp:add(nn.Tanh())
mlp:add(nn.Linear(HUs, outputs))
```

Loss function

We choose the Mean Squared Error criterion:

```
criterion = nn.MSECriterion()
```

Training

We create data *on the fly* and feed it to the neural network.

```
for i = 1,2500 do
  -- random sample
  local input= torch.randn(2);     -- normally distributed example
in 2d
  local output= torch.Tensor(1);
  if input[1]*input[2] > 0 then  -- calculate label for XOR
function
    output[1] = -1
  else
    output[1] = 1
  end

  -- feed it to the neural network and the criterion
  criterion:forward(mlp:forward(input), output)

  -- train over this example in 3 steps
  -- (1) zero the accumulation of the gradients
  mlp:zeroGradParameters()
  -- (2) accumulate gradients
  mlp:backward(input, criterion:backward(mlp.output, output))
  -- (3) update parameters with a 0.01 learning rate
  mlp:updateParameters(0.01)
end
```

Test the network

```
x = torch.Tensor(2)
x[1] =  0.5; x[2] =  0.5; print(mlp:forward(x))
x[1] =  0.5; x[2] = -0.5; print(mlp:forward(x))
x[1] = -0.5; x[2] =  0.5; print(mlp:forward(x))
x[1] = -0.5; x[2] = -0.5; print(mlp:forward(x))
```

You should see something like:

```
> x = torch.Tensor(2)
> x[1] =  0.5; x[2] =  0.5; print(mlp:forward(x))

-0.6140
[torch.Tensor of dimension 1]
```

```
> x[1] =  0.5; x[2] = -0.5; print(mlp:forward(x))

 0.8878
[torch.Tensor of dimension 1]

> x[1] = -0.5; x[2] =  0.5; print(mlp:forward(x))

 0.8548
[torch.Tensor of dimension 1]

> x[1] = -0.5; x[2] = -0.5; print(mlp:forward(x))

-0.5498
[torch.Tensor of dimension 1]
```

StochasticGradient

`StochasticGradient` is a high-level class for training neural networks, using a stochastic gradient
algorithm. This class is serializable.

StochasticGradient(module, criterion)

Create a `StochasticGradient` class, using the given Module and Criterion.
The class contains several parameters you might want to set after initialization.

train(dataset)

Train the module and criterion given in the
constructor over `dataset`, using the
internal parameters.

StochasticGradient expect as a `dataset` an object which implements the operator `dataset[index]` and implements the method `dataset:size()`. The `size()` methods returns the number of examples and `dataset[i]` has to return the i-th example.

An `example` has to be an object which implements the operator `example[field]`, where `field` might take the value `1` (input features) or `2` (corresponding label which will be given to the criterion).
The input is usually a Tensor (except if you use special kind of gradient modules, like table layers). The label type depends of the criterion.
For example, the MSECriterion expects a Tensor, but the ClassNLLCriterion except a integer number (the class).

Such a dataset is easily constructed by using Lua tables, but it could any `C` object for example, as long as required operators/methods are implemented.
See an example.

Parameters

`StochasticGradient` has several field which have an impact on a call to train().

- `learningRate` : This is the learning rate used during training. The update of the parameters will be `parameters = parameters - learningRate * parameters_gradient`. Default value is `0.01`.
- `learningRateDecay` : The learning rate decay. If non-zero, the learning rate (note: the field learningRate will not change value) will be computed after each iteration (pass over the dataset) with: `current_learning_rate =learningRate / (1 + iteration * learningRateDecay)`
- `maxIteration` : The maximum number of iteration (passes over the dataset). Default is `25`.
- `shuffleIndices` : Boolean which says if the examples will be randomly sampled or not. Default is `true`. If `false`, the examples will be taken in the order of the dataset.
- `hookExample` : A possible hook function which will be called (if non-nil) during training after each example forwarded and backwarded through the network. The function takes `(self, example)` as parameters. Default is `nil`.
- `hookIteration` : A possible hook function which will be called (if non-nil) during training after a complete pass over the dataset. The function takes `(self, iteration, currentError)` as parameters. Default is `nil`.

Example of training using StochasticGradient

We show an example here on a classical XOR problem.

Dataset

We first need to create a dataset, following the conventions described in StochasticGradient.

```
dataset={};
function dataset:size() return 100 end -- 100 examples
for i=1,dataset:size() do
  local input = torch.randn(2);     -- normally distributed example
in 2d
  local output = torch.Tensor(1);
  if input[1]*input[2]>0 then     -- calculate label for XOR
function
    output[1] = -1;
  else
    output[1] = 1
  end
  dataset[i] = {input, output}
end
```

Neural Network

We create a simple neural network with one hidden layer.

```
require "nn"
mlp = nn.Sequential();  -- make a multi-layer perceptron
inputs = 2; outputs = 1; HUs = 20; -- parameters
mlp:add(nn.Linear(inputs, HUs))
mlp:add(nn.Tanh())
mlp:add(nn.Linear(HUs, outputs))
```

Training

We choose the Mean Squared Error criterion and train the dataset.

```
criterion = nn.MSECriterion()
trainer = nn.StochasticGradient(mlp, criterion)
trainer.learningRate = 0.01
trainer:train(dataset)
```

Test the network

```
x = torch.Tensor(2)
x[1] =  0.5; x[2] =  0.5; print(mlp:forward(x))
x[1] =  0.5; x[2] = -0.5; print(mlp:forward(x))
x[1] = -0.5; x[2] =  0.5; print(mlp:forward(x))
x[1] = -0.5; x[2] = -0.5; print(mlp:forward(x))
```

You should see something like:

```
> x = torch.Tensor(2)
> x[1] =  0.5; x[2] =  0.5; print(mlp:forward(x))

-0.3490
[torch.Tensor of dimension 1]

> x[1] =  0.5; x[2] = -0.5; print(mlp:forward(x))

 1.0561
[torch.Tensor of dimension 1]

> x[1] = -0.5; x[2] =  0.5; print(mlp:forward(x))

 0.8640
[torch.Tensor of dimension 1]

> x[1] = -0.5; x[2] = -0.5; print(mlp:forward(x))

-0.2941
[torch.Tensor of dimension 1]
```

Using optim to train a network

`optim` is a powerful module, that provides multiple optimization algorithms.

Testing

For those who want to implement their own modules, we suggest using the `nn.Jacobian` class for testing the derivatives of their class, together with the torch.Tester class. The sources of `nn` package contains sufficiently many examples of such tests.

nn.Jacobian

testJacobian(module, input, minval, maxval, perturbation)

Test the jacobian of a module w.r.t. to its input.

`module` takes as its input a random tensor shaped the same as `input`.
`minval` and `maxval` specify the range of the random tensor ([-2, 2] by default).
`perturbation` is used as finite difference (1e-6 by default).

Returns the L-inf distance between the jacobian computed by backpropagation and by finite difference.

testJacobianParameters(module, input, param, dparam, minval, maxval, perturbation)

Test the jacobian of a module w.r.t. its parameters (instead of its input).

The input and parameters of `module` are random tensors shaped the same as `input` and `param`.
`minval` and `maxval` specify the range of the random tensors ([-2, 2] by default).
`dparam` points to the gradient w.r.t. parameters.
`perturbation` is used as finite difference (1e-6 by default).

Returns the L-inf distance between the jacobian computed by backpropagation and by finite difference.

testJacobianUpdateParameters(module, input, param, minval, maxval, perturbation)

Test the amount of update of a module to its parameters.

The input and parameters of `module` are random tensors shaped the same as `input` and `param`.
`minval` and `maxval` specify the range of the random tensors ([-2, 2] by default).
`perturbation` is used as finite difference (1e-6 by default).

Returns the L-inf distance between the update computed by backpropagation and by finite difference.

forward(module, input, param, perturbation)

Compute the jacobian by finite difference.

`module` has parameters `param` and input `input`.
If provided, `param` is regarded as independent variables, otherwise `input` is the independent variables.
`perturbation` is used as finite difference (1e-6 by default).

Returns the jacobian computed by finite difference.

backward(module, input, param, dparam)

Compute the jacobian by backpropagation.

`module` has parameters `param` and input `input`.
If provided, `param` is regarded as independent variables, otherwise `input` is the independent variables.
`dparam` is the gradient w.r.t. parameters, it must present as long as `param` is present.

Returns the jacobian computed by backpropagation.

Overview

Most optimization algorithms have the following interface:

```
x*, {f}, ... = optim.method(opfunc, x[, config][, state])
```

where:

- `opfunc` : a user-defined closure that respects this API: `f, df/dx = func(x)`
- `x` : the current parameter vector (a 1D `Tensor`)
- `config` : a table of parameters, dependent upon the algorithm
- `state` : a table of state variables, if `nil` , `config` will contain the state
- `x*` : the new parameter vector that minimizes `f, x* = argmin_x f(x)`
- `{f}` : a table of all `f` values, in the order they've been evaluated (for some simple algorithms, like SGD, `#f == 1`)

Example

The state table is used to hold the state of the algorithm.
It's usually initialized once, by the user, and then passed to the optim function as a black box.
Example:

```
config = {
   learningRate = 1e-3,
   momentum = 0.5
}

for i, sample in ipairs(training_samples) do
    local func = function(x)
       -- define eval function
       return f, df_dx
    end
    optim.sgd(func, x, config)
end
```

Training using optim

`optim` is a quite general optimizer, for minimizing any function with respect to a set of parameters.
In our case, our function will be the loss of our network, given an input, and a set of weights.
The goal of training a neural net is to optimize the weights to give the lowest loss over our validation set, by using the training set as a proxy.
So, we are going to use optim to minimize the loss with respect to the weights, over our training set.

To illustrate all the steps required, we will go over a simple example, where we will train a neural network on the classical XOR problem.
We will feed the data to `optim` in minibatches (we will use here just one minibatch), breaking your training set into chucks, and feed each minibatch to `optim`, one by one.

We need to give `optim` a function that will output the loss and the derivative of the loss with respect to the
weights, given the current weights, as a function parameter.
The function will have access to our training minibatch, and use this to calculate the loss, for this minibatch.
Typically, the function would be defined inside our loop over batches, and therefore have access to the current minibatch data.

Neural Network

We create a simple neural network with one hidden layer.

```
require 'nn'

model = nn.Sequential()  -- make a multi-layer perceptron
inputs = 2; outputs = 1; HUs = 20 -- parameters
model:add(nn.Linear(inputs, HUs))
model:add(nn.Tanh())
model:add(nn.Linear(HUs, outputs))
```

> If we would like to train on GPU, then we need to shipt the model to *device memory* by typing `model:cuda()` after having issued `require 'cunn'`.

Criterion

We choose the *Mean Squared Error* loss `Criterion` :

```
criterion = nn.MSECriterion()
```

We are using an `nn.MSECriterion` because we are training on a regression task, predicting contiguous (real) target value, from `-1` to `+1` .
For a classification task, with more than two classes, we would add an `nn.LogSoftMax` layer to the end of our network, and use a `nn.ClassNLLCriterion` loss criterion.
Nevertheless, the XOR problem could be seen and a two classes classification task, associated to the `-1` and `+1` discrete outputs.

> If we would like to train on GPU, then we need to ship the `Criterion` to *device memory* by typing `criterion:cuda()` .

Data set

We will just create one minibatch of `128` examples.
In your own training, you'd want to break down your rather larger data set into multiple minibatches, of around `32` to `512` examples each.

```
batchSize = 128
batchInputs = torch.DoubleTensor(batchSize, inputs) -- or
CudaTensor for GPU training
batchLabels = torch.DoubleTensor(batchSize)         -- or
CudaTensor for GPU training

for i = 1, batchSize do
   local input = torch.randn(2)     -- normally distributed example
in 2d
   local label
   if input[1] * input[2] > 0 then  -- calculate label for XOR
function
      label = -1
   else
      label = 1
   end
```

```
      batchInputs[i]:copy(input)
      batchLabels[i] = label
   end
```

Flatten parameters

`optim` expects the parameters that are to be optimized, and their gradients, to be one-dimensional `Tensor`s.
But, our network model contains probably multiple modules, typically multiple convolutional layers, and each of these layers has their own `weight` and `bias` `Tensor`s.
How to handle this?

It is simple: we can call a standard method `:getParameters()`, that is defined for any network module.
When we call this method, the following magic will happen:

- a new `Tensor` will be created, large enough to hold all the `weight`s and `bias`es of the entire network model
- the model `weight` and `bias` `Tensor`s are replaced with views onto the new contiguous parameter `Tensor`
- and the exact same thing will happen for all the gradient `Tensor`s: replaced with views onto one single contiguous gradient `Tensor`

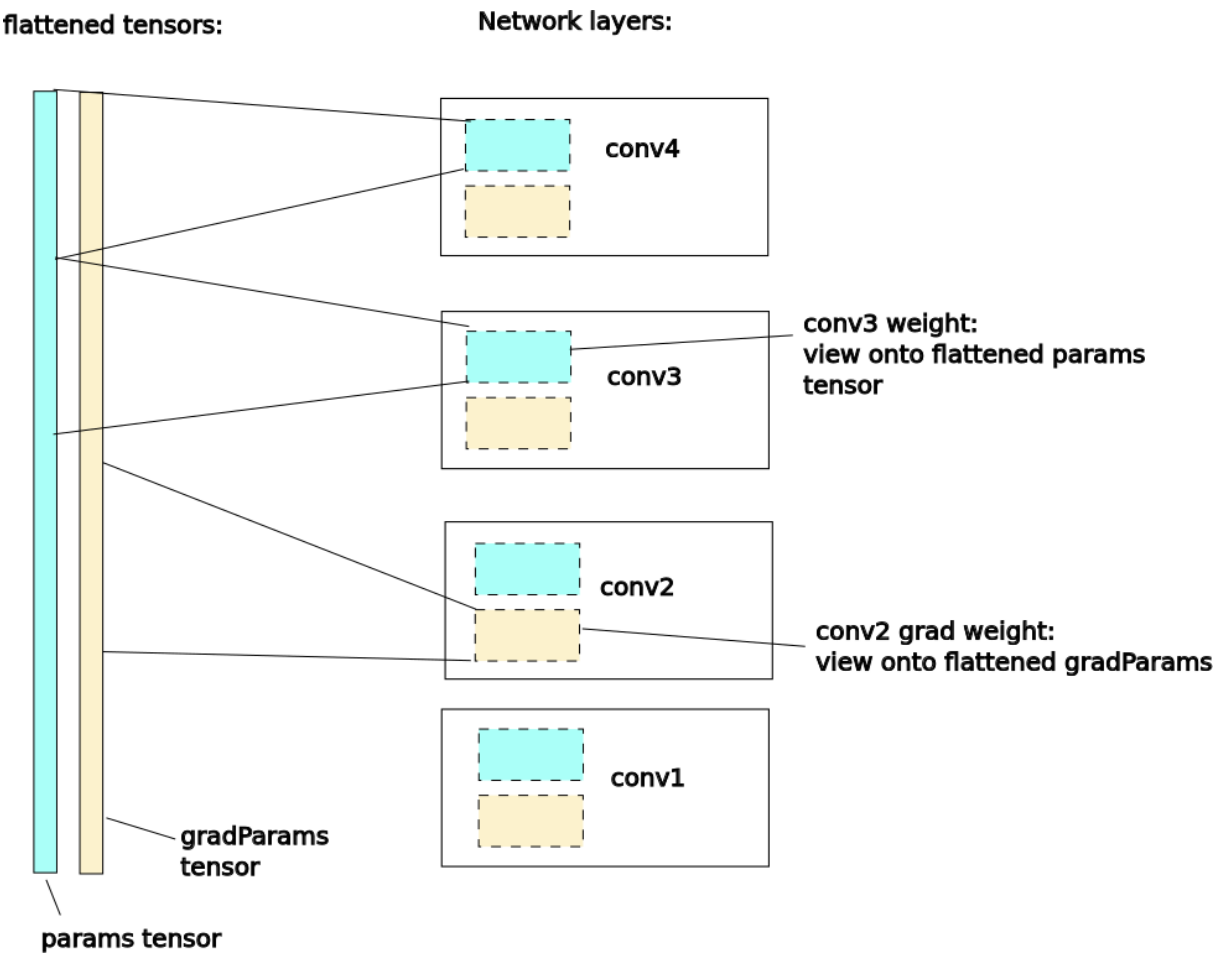
We can call this method as follows:

```
params, gradParams = model:getParameters()
```

These flattened `Tensor`s have the following characteristics:

- to `optim`, the parameters it needs to optimize are all contained in one single one-dimensional `Tensor`
- when `optim` optimizes the parameters in this large one-dimensional `Tensor`, it is implicitly optimizing the `weight`s and `bias`es in our network model, since those are now simply views onto this large one-dimensional parameter `Tensor`

It will look something like this:



> Note that flattening the parameters redefines the `weight` and `bias` `Tensor`s for all the network modules in our network model.
> Therefore, any pre-existing references to the original model layer `weight` and `bias` `Tensor`s will no longer point to the model `weight` and `bias` `Tensor`s, after flattening.

Training

Now that we have created our model, our training set, and prepared the flattened network parameters, we can train using `optim`.
`optim` provides various training algorithms.
We will use the stochastic gradient descent algorithm SGD.
We need to provide the learning rate, via an optimization state table:

```
local optimState = {learningRate = 0.01}
```

We define an evaluation function, inside our training loop, and use `optim.sgd` to train the system:

```
require 'optim'

for epoch = 1, 50 do
   -- local function we give to optim
   -- it takes current weights as input, and outputs the loss
   -- and the gradient of the loss with respect to the weights
   -- gradParams is calculated implicitly by calling 'backward',
   -- because the model's weight and bias gradient tensors
   -- are simply views onto gradParams
   function feval(params)
      gradParams:zero()

      local outputs = model:forward(batchInputs)
      local loss = criterion:forward(outputs, batchLabels)
      local dloss_doutputs = criterion:backward(outputs,
batchLabels)
      model:backward(batchInputs, dloss_doutputs)

      return loss, gradParams
   end
   optim.sgd(feval, params, optimState)
end
```

Test the network

For the prediction task, we will also typically use minibatches, although we can run prediction sample by sample too.
In this example, we will predict sample by sample.
To run prediction on a minibatch, simply pass in a tensor with one additional dimension, which represents the sample index.

```
x = torch.Tensor(2)
x[1] =  0.5; x[2] =  0.5; print(model:forward(x))
x[1] =  0.5; x[2] = -0.5; print(model:forward(x))
x[1] = -0.5; x[2] =  0.5; print(model:forward(x))
x[1] = -0.5; x[2] = -0.5; print(model:forward(x))
```

You should see something like:

```
> x = torch.Tensor(2)
> x[1] =  0.5; x[2] =  0.5; print(model:forward(x))

-0.3490
[torch.DoubleTensor of dimension 1]

> x[1] =  0.5; x[2] = -0.5; print(model:forward(x))

 1.0561
[torch.DoubleTensor of dimension 1]

> x[1] = -0.5; x[2] =  0.5; print(model:forward(x))

 0.8640
[torch.DoubleTensor of dimension 1]

> x[1] = -0.5; x[2] = -0.5; print(model:forward(x))

-0.2941
[torch.DoubleTensor of dimension 1]
```

If we were running on a GPU, we would probably want to predict using minibatches, because this will hide the latencies involved in transferring data from main memory to the GPU.
To predict on a minbatch, we could do something like:

```
x = torch.CudaTensor({
   { 0.5,  0.5},
   { 0.5, -0.5},
   {-0.5,  0.5},
   {-0.5, -0.5}
})
print(model:forward(x))
```

You should see something like:

```
> print(model:forward(x))
 -0.3490
  1.0561
  0.8640
 -0.2941
```

```
 [torch.CudaTensor of size 4]
```

That's it!
For minibatched prediction, the output tensor contains one value for each of our input data samples.

Optimization algorithms

The following algorithms are provided:

- *Stochastic Gradient Descent*
- *Averaged Stochastic Gradient Descent*
- *L-BFGS*
- *Congugate Gradients*
- *AdaDelta*
- *AdaGrad*
- *Adam*
- *AdaMax*
- *FISTA with backtracking line search*
- *Nesterov's Accelerated Gradient method*
- *RMSprop*
- *Rprop*
- *CMAES*

All these algorithms are designed to support batch optimization as well as stochastic optimization.
It's up to the user to construct an objective function that represents the batch, mini-batch, or single sample on which to evaluate the objective.

Some of these algorithms support a line search, which can be passed as a function (*L-BFGS*), whereas others only support a learning rate (*SGD*).

General interface:

```
x*, {f}, ... = optim.method(opfunc, x[, config][, state])
```

sgd(opfunc, x[, config][, state])

An implementation of *Stochastic Gradient Descent* (*SGD*).

Arguments:

- `opfunc` : a function that takes a single input `X` , the point of a evaluation, and returns

`f(X)` and `df/dX`
- `x` : the initial point
- `config` : a table with configuration parameters for the optimizer
- `config.learningRate` : learning rate
- `config.learningRateDecay` : learning rate decay
- `config.weightDecay` : weight decay
- `config.weightDecays` : vector of individual weight decays
- `config.momentum` : momentum
- `config.dampening` : dampening for momentum
- `config.nesterov` : enables Nesterov momentum
- `state` : a table describing the state of the optimizer; after each call the state is modified
- `state.learningRates` : vector of individual learning rates

Returns:

- `x*` : the new x vector
- `f(x)` : the function, evaluated before the update

asgd(opfunc, x[, config][, state])

An implementation of *Averaged Stochastic Gradient Descent* (*ASGD*):

```
x = (1 - lambda eta_t) x - eta_t df / dx(z, x)
a = a + mu_t [ x - a ]

eta_t = eta0 / (1 + lambda eta0 t) ^ 0.75
mu_t = 1 / max(1, t - t0)
```

Arguments:

- `opfunc` : a function that takes a single input `X` , the point of evaluation, and returns `f(X)` and `df/dX`
- `x` : the initial point
- `config` : a table with configuration parameters for the optimizer
- `config.eta0` : learning rate
- `config.lambda` : decay term
- `config.alpha` : power for eta update
- `config.t0` : point at which to start averaging

Returns:

- `x*` : the new x vector
- `f(x)` : the function, evaluated before the update
- `ax` : the averaged x vector

lbfgs(opfunc, x[, config][, state])

An implementation of *L-BFGS* that relies on a user-provided line search function (`state.lineSearch`).
If this function is not provided, then a simple learning rate is used to produce fixed size steps. Fixed size steps are much less costly than line searches, and can be useful for stochastic problems.

The learning rate is used even when a line search is provided.
This is also useful for large-scale stochastic problems, where opfunc is a noisy approximation of `f(x)`.
In that case, the learning rate allows a reduction of confidence in the step size.

Arguments:

- `opfunc` : a function that takes a single input `X`, the point of evaluation, and returns `f(X)` and `df/dX`
- `x` : the initial point
- `config` : a table with configuration parameters for the optimizer
- `config.maxIter` : Maximum number of iterations allowed
- `config.maxEval` : Maximum number of function evaluations
- `config.tolFun` : Termination tolerance on the first-order optimality
- `config.tolX` : Termination tol on progress in terms of func/param changes
- `config.lineSearch` : A line search function
- `config.learningRate` : If no line search provided, then a fixed step size is used

Returns:
* `x*` : the new `x` vector, at the optimal point
* `f` : a table of all function values:
* `f[1]` is the value of the function before any optimization and
* `f[#f]` is the final fully optimized value, at `x*`

cg(opfunc, x[, config][, state])

An implementation of the *Conjugate Gradient* method which is a rewrite of `minimize.m` written by Carl E. Rasmussen.
It is supposed to produce exactly same results (give or take numerical accuracy due to some changed order of operations).
You can compare the result on rosenbrock with `minimize.m`.

```
x, fx, c = minimize([0, 0]', 'rosenbrock', -25)
```

Note that we limit the number of function evaluations only, it seems much more important in practical use.

Arguments:

- `opfunc` : a function that takes a single input, the point of evaluation.
- `x` : the initial point
- `config` : a table with configuration parameters for the optimizer
- `config.maxEval` : max number of function evaluations
- `config.maxIter` : max number of iterations
- `state` : a table of parameters and temporary allocations.
- `state.df[0, 1, 2, 3]` : if you pass `Tensor` they will be used for temp storage
- `state.[s, x0]` : if you pass `Tensor` they will be used for temp storage

Returns:

- `x*` : the new `x` vector, at the optimal point
- `f` : a table of all function values where
  - `f[1]` is the value of the function before any optimization and
  - `f[#f]` is the final fully optimized value, at `x*`

adadelta(opfunc, x[, config][, state])

AdaDelta implementation for *SGD* http://arxiv.org/abs/1212.5701.

Arguments:

- `opfunc` : a function that takes a single input `X`, the point of evaluation, and returns `f(X)` and `df/dX`
- `x` : the initial point
- `config` : a table of hyper-parameters
- `config.rho` : interpolation parameter

- `config.eps` : for numerical stability
- `state` : a table describing the state of the optimizer; after each call the state is modified
- `state.paramVariance` : vector of temporal variances of parameters
- `state.accDelta` : vector of accummulated delta of gradients

Returns:

- `x*` : the new x vector
- `f(x)` : the function, evaluated before the update

adagrad(opfunc, x[, config][, state])

AdaGrad implementation for *SGD*.

Arguments:

- `opfunc` : a function that takes a single input `X` , the point of evaluation, and returns `f(X)` and `df/dX`
- `x` : the initial point
- `config` : a table with configuration parameters for the optimizer
- `config.learningRate` : learning rate
- `state` : a table describing the state of the optimizer; after each call the state is modified
- `state.paramVariance` : vector of temporal variances of parameters

Returns:

- `x*` : the new `x` vector
- `f(x)` : the function, evaluated before the update

adam(opfunc, x[, config][, state])

An implementation of *Adam* from http://arxiv.org/pdf/1412.6980.pdf.

Arguments:

- `opfunc` : a function that takes a single input `X` , the point of a evaluation, and returns `f(X)` and `df/dX`
- `x` : the initial point

- `config` : a table with configuration parameters for the optimizer
- `config.learningRate` : learning rate
- `config.learningRateDecay` : learning rate decay
- `config.beta1` : first moment coefficient
- `config.beta2` : second moment coefficient
- `config.epsilon` : for numerical stability
- `state` : a table describing the state of the optimizer; after each call the state is modified

Returns:

- `x*` : the new x vector
- `f(x)` : the function, evaluated before the update

adamax(opfunc, x[, config][, state])

An implementation of *AdaMax* http://arxiv.org/pdf/1412.6980.pdf.

Arguments:

- `opfunc` : a function that takes a single input `X` , the point of a evaluation, and returns `f(X)` and `df/dX`
- `x` : the initial point
- `config` : a table with configuration parameters for the optimizer
- `config.learningRate` : learning rate
- `config.beta1` : first moment coefficient
- `config.beta2` : second moment coefficient
- `config.epsilon` : for numerical stability
- `state` : a table describing the state of the optimizer; after each call the state is modified

Returns:

- `x*` : the new `x` vector
- `f(x)` : the function, evaluated before the update

FistaLS(f, g, pl, xinit[, params])

Fista with backtracking *Line Search*:

- `f` : smooth function
- `g` : non-smooth function
- `pl` : minimizer of intermediate problem Q(x, y)
- `xinit` : initial point
- `params` : table of parameters (**optional**)
- `params.L` : 1/(step size) for ISTA/FISTA iteration (0.1)
- `params.Lstep` : step size multiplier at each iteration (1.5)
- `params.maxiter` : max number of iterations (50)
- `params.maxline` : max number of line search iterations per iteration (20)
- `params.errthres` : Error thershold for convergence check (1e-4)
- `params.doFistaUpdate` : true : use FISTA, false: use ISTA (true)
- `params.verbose` : store each iteration solution and print detailed info (false)

On output, `params` will contain these additional fields that can be reused.
* `params.L` : last used L value will be written.

These are temporary storages needed by the algo and if the same params object is passed a second time, these same storages will be used without new allocation.
* `params.xkm` : previous iterarion point
* `params.y` : fista iteration
* `params.ply` : `ply = pl(y * 1/L grad(f))`

Returns the solution `x` and history of `{function evals, number of line search , ...}`.

Algorithm is published in http://epubs.siam.org/doi/abs/10.1137/080716542

nag(opfunc, x[, config][, state])

An implementation of *SGD* adapted with features of *Nesterov's Accelerated Gradient method*, based on the paper "On the Importance of Initialization and Momentum in Deep Learning" (Sutskever et. al., ICML 2013) http://www.cs.toronto.edu/~fritz/absps/momentum.pdf.

Arguments:

- `opfunc` : a function that takes a single input `X` , the point of evaluation, and returns `f(X)` and `df/dX`
- `x` : the initial point
- `config` : a table with configuration parameters for the optimizer
- `config.learningRate` : learning rate
- `config.learningRateDecay` : learning rate decay

- `config.weightDecay` : weight decay
- `config.momentum` : momentum
- `config.learningRates` : vector of individual learning rates

Returns:

- `x*` : the new `x` vector
- `f(x)` : the function, evaluated before the update

rmsprop(opfunc, x[, config][, state])

An implementation of *RMSprop*.

Arguments:

- `opfunc` : a function that takes a single input `X` , the point of a evaluation, and returns `f(X)` and `df/dX`
- `x` : the initial point
- `config` : a table with configuration parameters for the optimizer
- `config.learningRate` : learning rate
- `config.alpha` : smoothing constant
- `config.epsilon` : value with which to initialise m
- `state` : a table describing the state of the optimizer; after each call the state is modified
- `state.m` : leaky sum of squares of parameter gradients,
- `state.tmp` : and the square root (with epsilon smoothing)

Returns:

- `x*` : the new x vector
- `f(x)` : the function, evaluated before the update

rprop(opfunc, x[, config][, state])

A plain implementation of *Rprop* (Martin Riedmiller, Koray Kavukcuoglu 2013).

Arguments:

- `opfunc` : a function that takes a single input `X` , the point of evaluation, and returns

`f(X)` and `df/dX`
- `x` : the initial point
- `config` : a table with configuration parameters for the optimizer
- `config.stepsize` : initial step size, common to all components
- `config.etaplus` : multiplicative increase factor, > 1 (default 1.2)
- `config.etaminus` : multiplicative decrease factor, < 1 (default 0.5)
- `config.stepsizemax` : maximum stepsize allowed (default 50)
- `config.stepsizemin` : minimum stepsize allowed (default 1e-6)
- `config.niter` : number of iterations (default 1)

Returns:

- `x*` : the new x vector
- `f(x)` : the function, evaluated before the update

cmaes(opfunc, x[, config][, state])

An implementation of *CMAES* (*Covariance Matrix Adaptation Evolution Strategy*), ported from https://www.lri.fr/~hansen/barecmaes2.html.

CMAES is a stochastic, derivative-free method for heuristic global optimization of non-linear or non-convex continuous optimization problems.
Note that this method will on average take much more function evaluations to converge then a gradient based method.

Arguments:

If `state` is specified, then `config` is not used at all.
Otherwise `state` is `config`.

- `opfunc` : a function that takes a single input `X`, the point of evaluation, and returns `f(X)` and `df/dX`. Note that `df/dX` is not used and can be left 0
- `x` : the initial point
- `state` : a table describing the state of the optimizer; after each call the state is modified
- `state.sigma` : float, initial step-size (standard deviation in each coordinate)
- `state.maxEval` : int, maximal number of function evaluations
- `state.ftarget` : float, target function value
- `state.popsize` : population size. If this is left empty, `4 + int(3 * log(|x|))` will be used
- `state.ftarget` : stop if `fitness < ftarget`
- `state.verb_disp` : display info on console every verb_disp iteration, 0 for never

Returns:
* `x*` : the new `x` vector, at the optimal point
* `f` : a table of all function values:
* `f[1]` is the value of the function before any optimization and
* `f[#f]` is the final fully optimized value, at `x*`

Logger

`optim` provides also logging and live plotting capabilities via the `optim.Logger()` function.

Live logging is essential to monitor the *network accuracy* and *cost function* during training and testing, for spotting *under-* and *over-fitting*, for *early stopping* or just for monitoring the health of the current optimisation task.

Logging data

Let walk through an example to see how it works.

We start with initialising our logger connected to a text file `accuracy.log`.

```
logger = optim.Logger('accuracy.log')
```

We can decide to log on it, for example, *training* and *testing accuracies*.

```
logger:setNames{'Training acc.', 'Test acc.'}
```

And now we can populate our logger randomly.

```
for i = 1, 10 do
   trainAcc = math.random(0, 100)
   testAcc = math.random(0, 100)
   logger:add{trainAcc, testAcc}
end
```

We can `cat` `accuracy.log` and see what's in it.

```
Training acc.   Test acc.
 7.0000e+01      5.9000e+01
 7.6000e+01      8.0000e+00
 6.6000e+01      3.4000e+01
```

```
   7.4000e+01      4.3000e+01
   5.7000e+01      1.1000e+01
   5.0000e+00      9.8000e+01
   7.1000e+01      1.7000e+01
   9.8000e+01      2.7000e+01
   3.5000e+01      4.7000e+01
   6.8000e+01      5.8000e+01
```

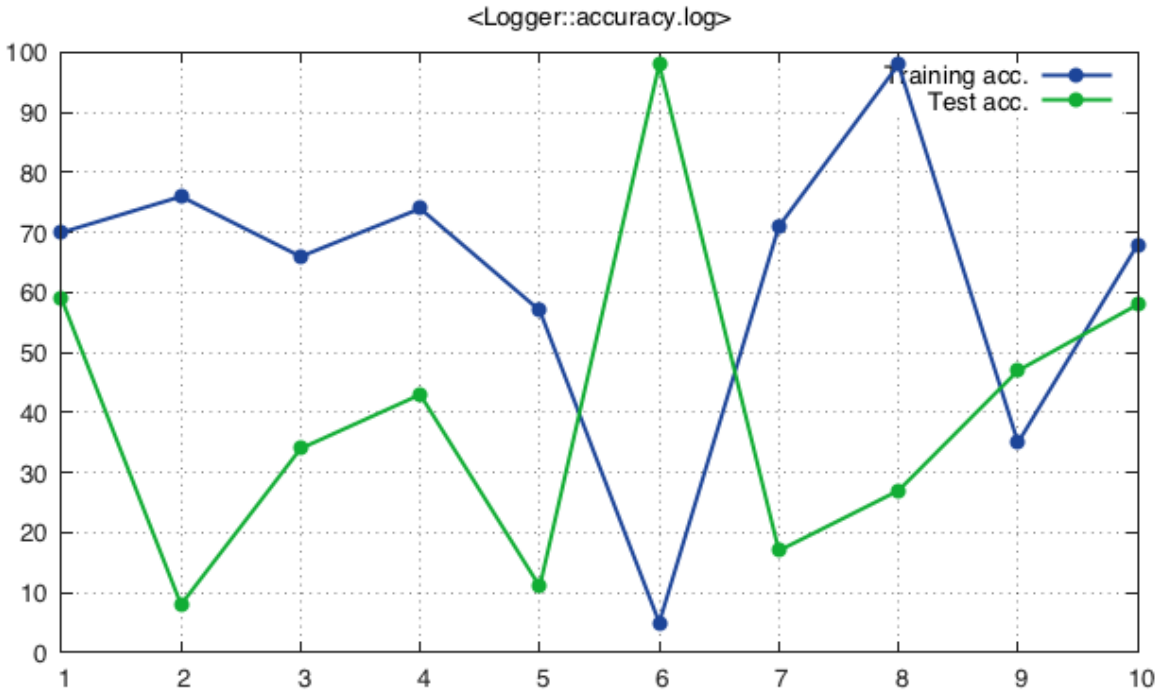
Visualising logs

OK, cool, but how can we actually see what's going on?

To have a better grasp of what's happening, we can plot our curves.
We need first to specify the plotting style, choosing from:

- `.` for dots
- `+` for points
- `-` for lines
- `+-` for points and lines
- `~` for using smoothed lines with cubic interpolation
- `|` for using boxes
- custom string, one can also pass custom strings to use full capability of gnuplot.

```
logger:style{'+-', '+-'}
logger:plot()
```



If we'd like an interactive visualisation, we can put the `logger:plot()` instruction within the `for` loop, and the chart will be updated at every iteration.

In case we'd like to prevent `gnuplot` to display the plots, we can set the option `logger:display(false)`.
In this way, plots will be saved but not displayed.
To restore the normal behaviour, use `logger:display(true)`.

We can set a logarithmic *y* axis with `logger:setlogscale(true)` and reset it with `logger:setlogscale(false)`.

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.

Manipulating Filenames

The following functions can be used
to manipulate filenames in a portable way
over multiple platforms.

paths.filep(path)

Return a boolean indicating whether `path`
refers to an existing file.

paths.dirp(path)

Return a boolean indicating whether `path`
refers to an existing directory.

paths.basename(path,[suffix])

Return the last path component of `path`
and optionally strip the suffix `suffix`.
This is similar to the well know shell command `"basename"`.

paths.dirname(path)

Return the name of directory containing file `path`.
This is similar to the well known shell command `"dirname"`.

paths.extname(path)

Return the extension of the `path` or nil if none is found.

paths.concat([path1,….,pathn])

Concatenates relative filenames.

First this function computes the full filename
of `path1` relative to the current directory.
Then it successively computes the full filenames
of arguments `path2` to `pathn` relative to
the filename returned for the previous argument.
Finally the last result is returned.

Calling this function without argument returns the
full name of the current directory.

paths.cwd()

Return the full path of the current working directory.

paths.execdir()

Return the name of the directory containing the
current Lua executable.
When the module `paths` is first loaded,
this information is used to relocate
the variables indicating
the location of the various Torch components.

paths.tmpname()

Return the name of a temporary file.
All the temporaty files whose name was obtained in this way
are removed when Lua exits.

This function should be preferred over `os.tmpname()`
because it makes sure that the files are removed on exit.
In addition, `os.tmpname()` under windows often returns filenames
for which the user has no permission to write.

Miscellaneous

paths.uname()

Returns up to three strings describing the operating system.
The first string is a system name, e.g., “Windows”, “Linux”, “Darwin”, “FreeBSD”, etc.
The second string is the network name of this computer.
The third string indicates the processor type.

paths.is_win()

Returns true if the operating system is Microsoft Windows.

paths.is_mac()

Returns true if the operating system is Mac OS X.

paths.getregistryvalue(key,subkey,value)

Query a value in the Windows registry value.
Causes an error on other systems.

paths.findprogram(progname,…)

Finds an executable program named “progname” and returns its full path.
If none is found, continue searching programs named after the following arguments and return the full path of the first match.
All the directories specified by the PATH variable are searched.
Under windows, this also searches the “App Path” registry entries.

paths.thisfile([arg])

Calling `paths.thisfile()` without argument
inside a lua file returns returns the full
pathname of the file from which it is called.
This function always returns `nil` when called
interactively.

Calling `paths.thisfile(arg)` with a string argument `arg`
returns the full pathname of the file `arg` relative
to the directory containing the file from which
function `paths.thisfile` is invoked. This is useful,
for instance, to locate files located in the same
directory as a lua script.

paths.dofile(filename)

This function is similar to the standard Lua function `dofile`
but interprets `filename` relative to the directory containing
the file that contains the call to `paths.dofile`,
or to the current directory when `paths.dofile` is
called interactively.

rnn: recurrent neural networks

This is a Recurrent Neural Network library that extends Torch's nn.
You can use it to build RNNs, LSTMs, GRUs, BRNNs, BLSTMs, and so forth and so on.
This library includes documentation for the following objects:

Modules that consider successive calls to `forward` as different time-steps in a sequence :
* AbstractRecurrent : an abstract class inherited by Recurrent and LSTM;
* Recurrent : a generalized recurrent neural network container;
* LSTM : a vanilla Long-Short Term Memory module;
* FastLSTM : a faster LSTM with optional support for batch normalization;
* GRU : Gated Recurrent Units module;
* Recursor : decorates a module to make it conform to the AbstractRecurrent interface;
* Recurrence : decorates a module that outputs `output(t)` given `{input(t), output(t-1)}` ;
* NormStabilizer : implements norm-stabilization criterion (add this module between RNNs);

Modules that `forward` entire sequences through a decorated `AbstractRecurrent` instance :
* AbstractSequencer : an abstract class inherited by Sequencer, Repeater, RecurrentAttention, etc.;
* Sequencer : applies an encapsulated module to all elements in an input sequence (Tensor or Table);
* SeqLSTM : a very fast version of `nn.Sequencer(nn.FastLSTM)` where the `input` and `output` are tensors;
* SeqLSTMP : `SeqLSTM` with a projection layer;
* SeqGRU : a very fast version of `nn.Sequencer(nn.GRU)` where the `input` and `output` are tensors;
* SeqBRNN : Bidirectional RNN based on SeqLSTM;
* BiSequencer : used for implementing Bidirectional RNNs and LSTMs;
* BiSequencerLM : used for implementing Bidirectional RNNs and LSTMs for language models;
* Repeater : repeatedly applies the same input to an AbstractRecurrent instance;
* RecurrentAttention : a generalized attention model for REINFORCE modules;

Miscellaneous modules and criterions :
* MaskZero : zeroes the `output` and `gradOutput` rows of the decorated module for commensurate `input` rows which are tensors of zeros;
* TrimZero : same behavior as `MaskZero` , but more efficient when `input` contains lots zero-masked rows;
* LookupTableMaskZero : extends `nn.LookupTable` to support zero indexes for padding. Zero indexes are forwarded as tensors of zeros;

* MaskZeroCriterion : zeros the `gradInput` and `err` rows of the decorated criterion for commensurate `input` rows which are tensors of zeros;
* SeqReverseSequence : reverses an input sequence on a specific dimension;

Criterions used for handling sequential inputs and targets :
* SequencerCriterion : sequentially applies the same criterion to a sequence of inputs and targets (Tensor or Table).
* RepeaterCriterion : repeatedly applies the same criterion with the same target on a sequence.

Examples

The following are example training scripts using this package :

- RNN/LSTM/GRU for Penn Tree Bank dataset;
- Noise Contrastive Estimate for training multi-layer SeqLSTM language models on the Google Billion Words dataset. The example uses MaskZero to train independent variable length sequences using the NCEModule and NCECriterion. This script is our fastest yet boasting speeds of 20,000 words/second (on NVIDIA Titan X) with a 2-layer LSTM having 250 hidden units, a batchsize of 128 and sequence length of a 100. Note that you will need to have Torch installed with Lua instead of LuaJIT;
- Recurrent Model for Visual Attention for the MNIST dataset;
- Encoder-Decoder LSTM shows you how to couple encoder and decoder `LSTMs` for sequence-to-sequence networks;
- Simple Recurrent Network shows a simple example for building and training a simple recurrent neural network;
- Simple Sequencer Network is a version of the above script that uses the Sequencer to decorate the `rnn` instead;
- Sequence to One demonstrates how to do many to one sequence learning as is the case for sentiment analysis;
- Multivariate Time Series demonstrates how train a simple RNN to do multi-variate time-series predication.

External Resources

- rnn-benchmarks : benchmarks comparing Torch (using this library), Theano and TensorFlow.
- Harvard Jupyter Notebook Tutorial : an in-depth tutorial for how to use the Element-Research rnn package by Harvard University;

- dpnn : this is a dependency of the **rnn** package. It contains useful nn extensions, modules and criterions;
- dataload : a collection of torch dataset loaders;
- RNN/LSTM/BRNN/BLSTM training script for Penn Tree Bank or Google Billion Words datasets;
- A brief (1 hours) overview of Torch7, which includes some details about the **rnn** packages (at the end), is available via this NVIDIA GTC Webinar video. In any case, this presentation gives a nice overview of Logistic Regression, Multi-Layer Perceptrons, Convolutional Neural Networks and Recurrent Neural Networks using Torch7;
- Sequence to Sequence mapping using encoder-decoder RNNs : a complete training example using synthetic data.
- ConvLSTM is a repository for training a Spatio-temporal video autoencoder with differentiable memory.
- An time series example for univariate timeseries prediction.

Citation

If you use **rnn** in your work, we'd really appreciate it if you could cite the following paper:

Léonard, Nicholas, Sagar Waghmare, Yang Wang, and Jin-Hwa Kim. rnn: Recurrent Library for Torch. arXiv preprint arXiv:1511.07889 (2015).

Any significant contributor to the library will also get added as an author to the paper.
A significant contributor
is anyone who added at least 300 lines of code to the library.

Troubleshooting

Most issues can be resolved by updating the various dependencies:

```
luarocks install torch
luarocks install nn
luarocks install dpnn
luarocks install torchx
```

If you are using CUDA :

```
luarocks install cutorch
luarocks install cunn
luarocks install cunnx
```

And don't forget to update this package :

```
luarocks install rnn
```

If that doesn't fix it, open and issue on github.

AbstractRecurrent

An abstract class inherited by Recurrent, LSTM and GRU.
The constructor takes a single argument :

```
rnn = nn.AbstractRecurrent([rho])
```

Argument `rho` is the maximum number of steps to backpropagate through time (BPTT).
Sub-classes can set this to a large number like 99999 (the default) if they want to backpropagate through
the entire sequence whatever its length. Setting lower values of rho are
useful when long sequences are forward propagated, but we only whish to
backpropagate through the last `rho` steps, which means that the remainder
of the sequence doesn't need to be stored (so no additional cost).

[recurrentModule] getStepModule(step)

Returns a module for time-step `step`. This is used internally by sub-classes
to obtain copies of the internal `recurrentModule`. These copies share
`parameters` and `gradParameters` but each have their own `output`, `gradInput`
and any other intermediate states.

setOutputStep(step)

This is a method reserved for internal use by Recursor
when doing backward propagation. It sets the object's `output` attribute
to point to the output at time-step `step`.
This method was introduced to solve a very annoying bug.

maskZero(nInputDim)

Decorates the internal `recurrentModule` with MaskZero.
The `output` Tensor (or table thereof) of the `recurrentModule`
will have each row (i.e. samples) zeroed when the commensurate row of the `input`
is a tensor of zeros.

The `nInputDim` argument must specify the number of non-batch dims
in the first Tensor of the `input`. In the case of an `input` table,
the first Tensor is the first one encountered when doing a depth-first search.

Calling this method makes it possible to pad sequences with different lengths in the same batch
with zero vectors.

When a sample time-step is masked (i.e. `input` is a row of zeros), then
the hidden state is effectively reset (i.e. forgotten) for the next non-mask time-step.
In other words, it is possible seperate unrelated sequences with a masked element.

trimZero(nInputDim)

Decorates the internal `recurrentModule` with TrimZero.

[output] updateOutput(input)

Forward propagates the input for the current step. The outputs or intermediate
states of the previous steps are used recurrently. This is transparent to the
caller as the previous outputs and intermediate states are memorized. This
method also increments the `step` attribute by 1.

updateGradInput(input, gradOutput)

Like `backward`, this method should be called in the reverse order of `forward` calls used to propagate a sequence. So for example :

```
rnn = nn.LSTM(10, 10) -- AbstractRecurrent instance
local outputs = {}
for i=1,nStep do -- forward propagate sequence
   outputs[i] = rnn:forward(inputs[i])
end

for i=nStep,1,-1 do -- backward propagate sequence in reverse order
   gradInputs[i] = rnn:backward(inputs[i], gradOutputs[i])
end

rnn:forget()
```

The reverse order implements backpropagation through time (BPTT).

accGradParameters(input, gradOutput, scale)

Like `updateGradInput`, but for accumulating gradients w.r.t. parameters.

recycle(offset)

This method goes hand in hand with `forget`. It is useful when the current time-step is greater than `rho`, at which point it starts recycling the oldest `recurrentModule` `sharedClones`, such that they can be reused for storing the next step. This `offset` is used for modules like `nn.Recurrent` that use a different module for the first step. Default offset is 0.

forget(offset)

This method brings back all states to the start of the sequence buffers, i.e. it forgets the current sequence. It also resets the `step` attribute to 1. It is highly recommended to call `forget` after each parameter update. Otherwise, the previous state will be used to activate the next, which will often lead to instability. This is caused by the previous state being

the result of now changed parameters. It is also good practice to call `forget` at the start of each new sequence.

maxBPTTstep(rho)

This method sets the maximum number of time-steps for which to perform backpropagation through time (BPTT). So say you set this to `rho = 3` time-steps, feed-forward for 4 steps, and then backpropgate, only the last 3 steps will be used for the backpropagation. If your AbstractRecurrent instance is wrapped by a Sequencer, this will be handled auto-magically by the Sequencer. Otherwise, setting this value to a large value (i.e. 9999999), is good for most, if not all, cases.

backwardOnline()

This method was deprecated Jan 6, 2016.
Since then, by default, `AbstractRecurrent` instances use the backwardOnline behaviour.
See updateGradInput for details.

training()

In training mode, the network remembers all previous `rho` (number of time-steps) states. This is necessary for BPTT.

evaluate()

During evaluation, since their is no need to perform BPTT at a later time, only the previous step is remembered. This is very efficient memory-wise, such that evaluation can be performed using potentially infinite-length sequence.

Recurrent

References :
* A. Sutsekever Thesis Sec. 2.5 and 2.8
* B. Mikolov Thesis Sec. 3.2 and 3.3
* C. RNN and Backpropagation Guide

A composite Module for implementing Recurrent Neural Networks (RNN), excluding the output layer.

The `nn.Recurrent(start, input, feedback, [transfer, rho, merge])` constructor takes 6 arguments:
* `start` : the size of the output (excluding the batch dimension), or a Module that will be inserted between the `input` Module and `transfer` module during the first step of the propagation. When `start` is a size (a number or `torch.LongTensor`), then this *start* Module will be initialized as `nn.Add(start)` (see Ref. A).
* `input` : a Module that processes input Tensors (or Tables). Output must be of same size as `start` (or its output in the case of a `start` Module), and same size as the output of the `feedback` Module.
* `feedback` : a Module that feedbacks the previous output Tensor (or Tables) up to the `merge` module.
* `merge` : a table Module that merges the outputs of the `input` and `feedback` Module before being forwarded through the `transfer` Module.
* `transfer` : a non-linear Module used to process the output of the `merge` module, or in the case of the first step, the output of the `start` Module.
* `rho` : the maximum amount of backpropagation steps to take back in time. Limits the number of previous steps kept in memory. Due to the vanishing gradients effect, references A and B recommend `rho = 5` (or lower). Defaults to 99999.

An RNN is used to process a sequence of inputs.
Each step in the sequence should be propagated by its own `forward` (and `backward`), one `input` (and `gradOutput`) at a time.
Each call to `forward` keeps a log of the intermediate states (the `input` and many `Module.outputs`)
and increments the `step` attribute by 1.
Method `backward` must be called in reverse order of the sequence of calls to `forward` in order to backpropgate through time (BPTT). This reverse order is necessary
to return a `gradInput` for each call to `forward`.

The `step` attribute is only reset to 1 when a call to the `forget` method is made.
In which case, the Module is ready to process the next sequence (or batch thereof).
Note that the longer the sequence, the more memory that will be required to store all the `output` and `gradInput` states (one for each time step).

To use this module with batches, we suggest using different
sequences of the same size within a batch and calling `updateParameters`

every `rho` steps and `forget` at the end of the sequence.

Note that calling the `evaluate` method turns off long-term memory;
the RNN will only remember the previous output. This allows the RNN
to handle long sequences without allocating any additional memory.

For a simple concise example of how to make use of this module, please consult the
simple-recurrent-network.lua
training script.

Decorate it with a Sequencer

Note that any `AbstractRecurrent` instance can be decorated with a Sequencer
such that an entire sequence (a table) can be presented with a single `forward/backward`
call.
This is actually the recommended approach as it allows RNNs to be stacked and makes the
rnn conform to the Module interface, i.e. each call to `forward` can be
followed by its own immediate call to `backward` as each `input` to the
model is an entire sequence, i.e. a table of tensors where each tensor represents
a time-step.

```
seq = nn.Sequencer(module)
```

The simple-sequencer-network.lua training script
is equivalent to the above mentionned simple-recurrent-network.lua
script, except that it decorates the `rnn` with a `Sequencer` which takes
a table of `inputs` and `gradOutputs` (the sequence for that batch).
This lets the `Sequencer` handle the looping over the sequence.

You should only think about using the `AbstractRecurrent` modules without
a `Sequencer` if you intend to use it for real-time prediction.
Actually, you can even use an `AbstractRecurrent` instance decorated by a `Sequencer`
for real time prediction by calling `Sequencer:remember()` and presenting each
time-step `input` as `{input}`.

Other decorators can be used such as the Repeater or RecurrentAttention.
The `Sequencer` is only the most common one.

LSTM

References :
* A. Speech Recognition with Deep Recurrent Neural Networks
* B. Long-Short Term Memory
* C. LSTM: A Search Space Odyssey
* D. nngraph LSTM implementation on github

This is an implementation of a vanilla Long-Short Term Memory module.
We used Ref. A's LSTM as a blueprint for this module as it was the most concise.
Yet it is also the vanilla LSTM described in Ref. C.

The `nn.LSTM(inputSize, outputSize, [rho])` constructor takes 3 arguments:
* `inputSize` : a number specifying the size of the input;
* `outputSize` : a number specifying the size of the output;
* `rho` : the maximum amount of backpropagation steps to take back in time. Limits the number of previous steps kept in memory. Defaults to 9999.

The actual implementation corresponds to the following algorithm:

```
i[t] = σ(W[x->i]x[t] + W[h->i]h[t-1] + W[c->i]c[t-1] + b[1->i])
(1)
f[t] = σ(W[x->f]x[t] + W[h->f]h[t-1] + W[c->f]c[t-1] + b[1->f])
(2)
z[t] = tanh(W[x->c]x[t] + W[h->c]h[t-1] + b[1->c])
(3)
c[t] = f[t]c[t-1] + i[t]z[t]
(4)
o[t] = σ(W[x->o]x[t] + W[h->o]h[t-1] + W[c->o]c[t] + b[1->o])
(5)
h[t] = o[t]tanh(c[t])
(6)
```

where `W[s->q]` is the weight matrix from `s` to `q`, `t` indexes the time-step,
`b[1->q]` are the biases leading into `q`, `σ()` is `Sigmoid`, `x[t]` is the input,
`i[t]` is the input gate (eq. 1), `f[t]` is the forget gate (eq. 2),
`z[t]` is the input to the cell (which we call the hidden) (eq. 3),
`c[t]` is the cell (eq. 4), `o[t]` is the output gate (eq. 5),
and `h[t]` is the output of this module (eq. 6). Also note that the
weight matrices from cell to gate vectors are diagonal `W[c->s]`, where `s`
is `i`, `f`, or `o`.

As you can see, unlike Recurrent, this
implementation isn't generic enough that it can take arbitrary component Module

definitions at construction. However, the LSTM module can easily be adapted through inheritance by overriding the different factory methods :
* `buildGate` : builds generic gate that is used to implement the input, forget and output gates;
* `buildInputGate` : builds the input gate (eq. 1). Currently calls `buildGate` ;
* `buildForgetGate` : builds the forget gate (eq. 2). Currently calls `buildGate` ;
* `buildHidden` : builds the hidden (eq. 3);
* `buildCell` : builds the cell (eq. 4);
* `buildOutputGate` : builds the output gate (eq. 5). Currently calls `buildGate` ;
* `buildModel` : builds the actual LSTM model which is used internally (eq. 6).

Note that we recommend decorating the `LSTM` with a `Sequencer` (refer to this for details).

FastLSTM

A faster version of the LSTM.
Basically, the input, forget and output gates, as well as the hidden state are computed at one fellswoop.

Note that `FastLSTM` does not use peephole connections between cell and gates. The algorithm from `LSTM` changes as follows:

```
i[t] = σ(W[x->i]x[t] + W[h->i]h[t-1] + b[1->i])
(1)
f[t] = σ(W[x->f]x[t] + W[h->f]h[t-1] + b[1->f])
(2)
z[t] = tanh(W[x->c]x[t] + W[h->c]h[t-1] + b[1->c])
(3)
c[t] = f[t]c[t-1] + i[t]z[t]
(4)
o[t] = σ(W[x->o]x[t] + W[h->o]h[t-1] + b[1->o])
(5)
h[t] = o[t]tanh(c[t])
(6)
```

i.e. omitting the summands `W[c->i]c[t-1]` (eq. 1), `W[c->f]c[t-1]` (eq. 2), and `W[c->o]c[t]` (eq. 5).

usenngraph

This is a static attribute of the `FastLSTM` class. The default value is `false`.
Setting `usenngraph = true` will force all new instantiated instances of `FastLSTM` to use `nngraph`'s `nn.gModule` to build the internal `recurrentModule` which is cloned for each time-step.

Recurrent Batch Normalization

This extends the `FastLSTM` class to enable faster convergence during training by zero-centering the input-to-hidden and hidden-to-hidden transformations.
It reduces the internal covariate shift between time steps. It is an implementation of Cooijmans et. al.'s Recurrent Batch Normalization. The hidden-to-hidden transition of each LSTM cell is normalized according to

```
i[t] = σ(BN(W[x->i]x[t]) + BN(W[h->i]h[t-1]) + b[1->i])
(1)
f[t] = σ(BN(W[x->f]x[t]) + BN(W[h->f]h[t-1]) + b[1->f])
(2)
z[t] = tanh(BN(W[x->c]x[t]) + BN(W[h->c]h[t-1]) + b[1->c])
(3)
c[t] = f[t]c[t-1] + i[t]z[t]
(4)
o[t] = σ(BN(W[x->o]x[t]) + BN(W[h->o]h[t-1]) + b[1->o])
(5)
h[t] = o[t]tanh(c[t])
(6)
```

where the batch normalizing transform is:

$$BN(h; gamma, beta) = beta + gamma * \frac{hd - E(hd)}{sqrt(E(\sigma(hd) + eps))}$$

where `hd` is a vector of (pre)activations to be normalized, `gamma`, and `beta` are model parameters that determine the mean and standard deviation of the normalized activation. `eps` is a regularization hyperparameter to keep the division numerically stable and `E(hd)` and `E(σ(hd))` are the estimates of the mean and variance in the mini-batch respectively. The authors recommend initializing `gamma` to a small value and found 0.1 to be the value that did not cause vanishing gradients. `beta`, the shift parameter, is `null` by default.

To turn on batch normalization during training, do:

```
nn.FastLSTM.bn = true
lstm = nn.FastLSTM(inputsize, outputsize, [rho, eps, momentum,
affine]
```

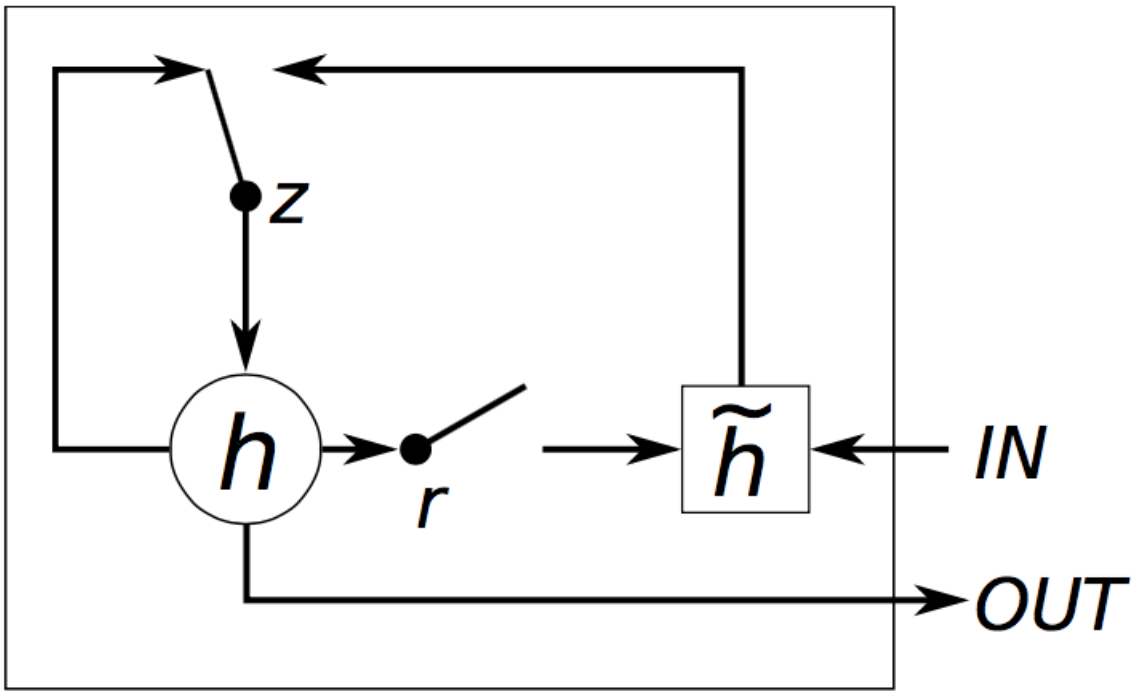
where `momentum` is same as `gamma` in the equation above (defaults to 0.1), `eps` is defined above and `affine` is a boolean whose state determines if the learnable affine transform is turned off (`false`) or on (`true` , the default).

GRU

References :
* A. Learning Phrase Representations Using RNN Encoder-Decoder For Statistical Machine Translation.
* B. Implementing a GRU/LSTM RNN with Python and Theano
* C. An Empirical Exploration of Recurrent Network Architectures
* D. Empirical Evaluation of Gated Recurrent Neural Networks on Sequence Modeling
* E. RnnDrop: A Novel Dropout for RNNs in ASR
* F. A Theoretically Grounded Application of Dropout in Recurrent Neural Networks

This is an implementation of Gated Recurrent Units module.

The `nn.GRU(inputSize, outputSize [,rho [,p [, mono]]])` constructor takes 3 arguments likewise `nn.LSTM` or 4 arguments for dropout:
* `inputSize` : a number specifying the size of the input;
* `outputSize` : a number specifying the size of the output;
* `rho` : the maximum amount of backpropagation steps to take back in time. Limits the number of previous steps kept in memory. Defaults to 9999;
* `p` : dropout probability for inner connections of GRUs.
* `mono` : Monotonic sample for dropouts inside GRUs. Only needed in a `TrimZero` + `BGRU` (p>0) situation.



The actual implementation corresponds to the following algorithm:

```
z[t] = σ(W[x->z]x[t] + W[s->z]s[t-1] + b[1->z])            (1)
r[t] = σ(W[x->r]x[t] + W[s->r]s[t-1] + b[1->r])            (2)
h[t] = tanh(W[x->h]x[t] + W[hr->c](s[t-1]r[t]) + b[1->h])  (3)
s[t] = (1-z[t])h[t] + z[t]s[t-1]                           (4)
```

where `W[s->q]` is the weight matrix from `s` to `q`, `t` indexes the time-step, `b[1->q]` are the biases leading into `q`, `σ()` is `Sigmoid`, `x[t]` is the input and `s[t]` is the output of the module (eq. 4). Note that unlike the LSTM, the GRU has no cells.

The GRU was benchmark on `PennTreeBank` dataset using recurrent-language-model.lua script.
It slightly outperfomed `FastLSTM`, however, since LSTMs have more parameters than GRUs, the dataset larger than `PennTreeBank` might change the performance result.
Don't be too hasty to judge on which one is the better of the two (see Ref. C and D).

```
               Memory   examples/s
FastLSTM         176M        16.5K
GRU               92M        15.8K
```

Memory is measured by the size of `dp.Experiment` save file. **examples/s** is measured by the training speed at 1 epoch, so, it may have a disk IO bias.

RNN dropout (see Ref. E and F) was benchmark on `PennTreeBank` dataset using `recurrent-language-model.lua` script, too. The details can be found in the script. In the benchmark, `GRU` utilizes a dropout after `LookupTable`, while `BGRU`, stands for Bayesian GRUs, uses dropouts on inner connections (naming as Ref. F), but not after `LookupTable`.

As Yarin Gal (Ref. F) mentioned, it is recommended that one may use `p = 0.25` for the first attempt.

Recursor

This module decorates a `module` to be used within an `AbstractSequencer` instance.
It does this by making the decorated module conform to the `AbstractRecurrent` interface, which like the `LSTM` and `Recurrent` classes, this class inherits.

```
rec = nn.Recursor(module[, rho])
```

For each successive call to `updateOutput` (i.e. `forward`), this decorator will create a `stepClone()` of the decorated `module`.
So for each time-step, it clones the `module`. Both the clone and original share parameters and gradients w.r.t. parameters. However, for modules that already conform to the `AbstractRecurrent` interface, the clone and original module are one and the same (i.e. no clone).

Examples :

Let's assume I want to stack two LSTMs. I could use two sequencers :

```
lstm = nn.Sequential()
   :add(nn.Sequencer(nn.LSTM(100,100)))
   :add(nn.Sequencer(nn.LSTM(100,100)))
```

Using a `Recursor`, I make the same model with a single `Sequencer` :

```
lstm = nn.Sequencer(
   nn.Recursor(
      nn.Sequential()
         :add(nn.LSTM(100,100))
         :add(nn.LSTM(100,100))
      )
   )
```

Actually, the `Sequencer` will wrap any non-`AbstractRecurrent` module automatically, so I could simplify this further to :

```
lstm = nn.Sequencer(
   nn.Sequential()
      :add(nn.LSTM(100,100))
      :add(nn.LSTM(100,100))
   )
```

I can also add a `Linear` between the two `LSTM`s. In this case, a `Linear` will be cloned (and have its parameters shared) for each time-step, while the `LSTM`s will do whatever cloning internally :

```
lstm = nn.Sequencer(
   nn.Sequential()
      :add(nn.LSTM(100,100))
      :add(nn.Linear(100,100))
      :add(nn.LSTM(100,100))
   )
```

`AbstractRecurrent` instances like `Recursor`, `Recurrent` and `LSTM` are expcted to manage time-steps internally. Non-`AbstractRecurrent` instances can be wrapped by a `Recursor` to have the same behavior.

Every call to `forward` on an `AbstractRecurrent` instance like `Recursor` will increment the `self.step` attribute by 1, using a shared parameter clone for each successive time-step (for a maximum of `rho` time-steps, which defaults to 9999999). In this way, `backward` can be called in reverse order of the `forward` calls to perform backpropagation through time (BPTT). Which is exactly what AbstractSequencer instances do internally.
The `backward` call, which is actually divided into calls to `updateGradInput` and `accGradParameters`, decrements by 1 the `self.udpateGradInputStep` and `self.accGradParametersStep`

respectively, starting at `self.step`.
Successive calls to `backward` will decrement these counters and use them to backpropagate through the appropriate internall step-wise shared-parameter clones.

Anyway, in most cases, you will not have to deal with the `Recursor` object directly as `AbstractSequencer` instances automatically decorate non-`AbstractRecurrent` instances
with a `Recursor` in their constructors.

For a concrete example of its use, please consult the simple-recurrent-network.lua training script for an example of its use.

Recurrence

A extremely general container for implementing pretty much any type of recurrence.

```
rnn = nn.Recurrence(recurrentModule, outputSize, nInputDim, [rho])
```

Unlike Recurrent, this module doesn't manage a separate
modules like `inputModule`, `startModule`, `mergeModule` and the like.
Instead, it only manages a single `recurrentModule`, which should
output a Tensor or table: `output(t)`
given an input table: `{input(t), output(t-1)}`.
Using a mix of `Recursor` (say, via `Sequencer`) with `Recurrence`, one can implement pretty much any type of recurrent neural network, including LSTMs and RNNs.

For the first step, the `Recurrence` forwards a Tensor (or table thereof) of zeros through the recurrent layer (like LSTM, unlike Recurrent).
So it needs to know the `outputSize`, which is either a number or `torch.LongStorage`, or table thereof. The batch dimension should be excluded from the `outputSize`. Instead, the size of the batch dimension (i.e. number of samples) will be extrapolated from the `input` using the `nInputDim` argument. For example, say that our input is a Tensor of size `4 x 3` where `4` is the number of samples, then `nInputDim` should be `1`.
As another example, if our input is a table of table [...] of tensors where the first tensor (depth first) is the same as in the previous example, then our `nInputDim` is also `1`.

As an example, let's use `Sequencer` and `Recurrence` to build a Simple RNN for language modeling:

```
rho = 5
hiddenSize = 10
outputSize = 5 -- num classes
nIndex = 10000

-- recurrent module
rm = nn.Sequential()
   :add(nn.ParallelTable()
      :add(nn.LookupTable(nIndex, hiddenSize))
      :add(nn.Linear(hiddenSize, hiddenSize)))
   :add(nn.CAddTable())
   :add(nn.Sigmoid())

rnn = nn.Sequencer(
   nn.Sequential()
      :add(nn.Recurrence(rm, hiddenSize, 1))
      :add(nn.Linear(hiddenSize, outputSize))
      :add(nn.LogSoftMax())
)
```

Note : We could very well reimplement the `LSTM` module using the newer `Recursor` and `Recurrent` modules, but that would mean breaking backwards compatibility for existing models saved on disk.

NormStabilizer

Ref. A : Regularizing RNNs by Stabilizing Activations

This module implements the norm-stabilization criterion:

```
ns = nn.NormStabilizer([beta])
```

This module regularizes the hidden states of RNNs by minimizing the difference between the L2-norms of consecutive steps. The cost function is defined as :

```
loss = beta * 1/T sum_t( ||h[t]|| - ||h[t-1]|| )^2
```

where `T` is the number of time-steps. Note that we do not divide the gradient by `T`

such that the chosen `beta` can scale to different sequence sizes without being changed.

The sole argument `beta` is defined in ref. A. Since we don't divide the gradients by
the number of time-steps, the default value of `beta=1` should be valid for most cases.

This module should be added between RNNs (or LSTMs or GRUs) to provide better
regularization of the hidden states.
For example :

```
local stepmodule = nn.Sequential()
   :add(nn.FastLSTM(10,10))
   :add(nn.NormStabilizer())
   :add(nn.FastLSTM(10,10))
   :add(nn.NormStabilizer())
local rnn = nn.Sequencer(stepmodule)
```

To use it with `SeqLSTM` you can do something like this :

```
local rnn = nn.Sequential()
   :add(nn.SeqLSTM(10,10))
   :add(nn.Sequencer(nn.NormStabilizer()))
   :add(nn.SeqLSTM(10,10))
   :add(nn.Sequencer(nn.NormStabilizer()))
```

AbstractSequencer

This abstract class implements a light interface shared by
subclasses like : `Sequencer` , `Repeater` , `RecurrentAttention` , `BiSequencer` and so on.

Sequencer

The `nn.Sequencer(module)` constructor takes a single argument, `module` , which is the
module
to be applied from left to right, on each element of the input sequence.

```
seq = nn.Sequencer(module)
```

This Module is a kind of decorator
used to abstract away the intricacies of `AbstractRecurrent` modules. While an `AbstractRecurrent` instance
requires that a sequence to be presented one input at a time, each with its own call to `forward` (and `backward`),
the `Sequencer` forwards an `input` sequence (a table) into an `output` sequence (a table of the same length).
It also takes care of calling `forget` on AbstractRecurrent instances.

Input/Output Format

The `Sequencer` requires inputs and outputs to be of shape `seqlen x batchsize x featsize`:

- `seqlen` is the number of time-steps that will be fed into the `Sequencer`.
- `batchsize` is the number of examples in the batch. Each example is its own independent sequence.
- `featsize` is the size of the remaining non-batch dimensions. So this could be `1` for language models, or `c x h x w` for convolutional models, etc.

Above is an example input sequence for a character level language model.
It has `seqlen` is 5 which means that it contains sequences of 5 time-steps.
The openning `{` and closing `}` illustrate that the time-steps are elements of a Lua table, although
it also accepts full Tensors of shape `seqlen x batchsize x featsize`.
The `batchsize` is 2 as their are two independent sequences: `{ H, E, L, L, O }` and `{ F, U, Z, Z, Y, }`.
The `featsize` is 1 as their is only one feature dimension per character and each such character is of size 1.
So the input in this case is a table of `seqlen` time-steps where each time-step is represented by a `batchsize x featsize` Tensor.

Above is another example of a sequence (input or output).
It has a `seqlen` of 4 time-steps.
The `batchsize` is again 2 which means there are two sequences.

The `featsize` is 3 as each time-step of each sequence has 3 variables.
So each time-step (element of the table) is represented again as a tensor
of size `batchsize x featsize`.
Note that while in both examples the `featsize` encodes one dimension,
it could encode more.

Example

For example, `rnn` : an instance of nn.AbstractRecurrent, can forward an `input` sequence one forward at a time:

```
input = {torch.randn(3,4), torch.randn(3,4), torch.randn(3,4)}
rnn:forward(input[1])
rnn:forward(input[2])
rnn:forward(input[3])
```

Equivalently, we can use a Sequencer to forward the entire `input` sequence at once:

```
seq = nn.Sequencer(rnn)
seq:forward(input)
```

We can also forward Tensors instead of Tables :

```
-- seqlen x batchsize x featsize
input = torch.randn(3,3,4)
seq:forward(input)
```

Details

The `Sequencer` can also take non-recurrent Modules (i.e. non-AbstractRecurrent instances) and apply it to each
input to produce an output table of the same length.
This is especially useful for processing variable length sequences (tables).

Internally, the `Sequencer` expects the decorated `module` to be an `AbstractRecurrent` instance. When this is not the case, the `module` is automatically decorated with a Recursor module, which makes it

conform to the `AbstractRecurrent` interface.

Note : this is due a recent update (27 Oct 2015), as before this `AbstractRecurrent` and and non-`AbstractRecurrent` instances needed to be decorated by their own `Sequencer`. The recent update, which introduced the `Recursor` decorator, allows a single `Sequencer` to wrap any type of module, `AbstractRecurrent`, non-`AbstractRecurrent` or a composite structure of both types. Nevertheless, existing code shouldn't be affected by the change.

For a concise example of its use, please consult the simple-sequencer-network.lua training script.

remember([mode])

When `mode='neither'` (the default behavior of the class), the Sequencer will additionally call forget before each call to `forward`.
When `mode='both'` (the default when calling this function), the Sequencer will never call forget.
In which case, it is up to the user to call `forget` between independent sequences.
This behavior is only applicable to decorated AbstractRecurrent `modules`.
Accepted values for argument `mode` are as follows :

- 'eval' only affects evaluation (recommended for RNNs)
- 'train' only affects training
- 'neither' affects neither training nor evaluation (default behavior of the class)
- 'both' affects both training and evaluation (recommended for LSTMs)

forget()

Calls the decorated AbstractRecurrent module's `forget` method.

SeqLSTM

This module is a faster version of `nn.Sequencer(nn.FastLSTM(inputsize, outputsize))` :

```
seqlstm = nn.SeqLSTM(inputsize, outputsize)
```

Each time-step is computed as follows (same as FastLSTM):

```
i[t] = σ(W[x->i]x[t] + W[h->i]h[t-1] + b[1->i])
(1)
f[t] = σ(W[x->f]x[t] + W[h->f]h[t-1] + b[1->f])
(2)
z[t] = tanh(W[x->c]x[t] + W[h->c]h[t-1] + b[1->c])
(3)
c[t] = f[t]c[t-1] + i[t]z[t]
(4)
o[t] = σ(W[x->o]x[t] + W[h->o]h[t-1] + b[1->o])
(5)
h[t] = o[t]tanh(c[t])
(6)
```

A notable difference is that this module expects the `input` and `gradOutput` to be tensors instead of tables. The default shape is `seqlen x batchsize x inputsize` for the `input` and `seqlen x batchsize x outputsize` for the `output` :

```
input = torch.randn(seqlen, batchsize, inputsize)
gradOutput = torch.randn(seqlen, batchsize, outputsize)

output = seqlstm:forward(input)
gradInput = seqlstm:backward(input, gradOutput)
```

Note that if you prefer to transpose the first two dimension (i.e. `batchsize x seqlen` instead of the default `seqlen x batchsize`)
you can set `seqlstm.batchfirst = true` following initialization.

For variable length sequences, set `seqlstm.maskzero = true` .
This is equivalent to calling `maskZero(1)` on a `FastLSTM` wrapped by a `Sequencer` :

```
fastlstm = nn.FastLSTM(inputsize, outputsize)
fastlstm:maskZero(1)
seqfastlstm = nn.Sequencer(fastlstm)
```

For `maskzero = true` , input sequences are expected to be seperated by tensor of zeros for a time step.

The `seqlstm:toFastLSTM()` method generates a FastLSTM instance initialized with the parameters
of the `seqlstm` instance. Note however that the resulting parameters will not be shared (nor can they ever be).

Like the `FastLSTM`, the `SeqLSTM` does not use peephole connections between cell and gates (see FastLSTM for details).

Like the `Sequencer`, the `SeqLSTM` provides a remember method.

Note that a `SeqLSTM` cannot replace `FastLSTM` in code that decorates it with a `AbstractSequencer` or `Recursor` as this would be equivalent to `Sequencer(Sequencer(FastLSTM))`.
You have been warned.

SeqLSTMP

References:
* A. LSTM RNN Architectures for Large Scale Acoustic Modeling
* B. Exploring the Limits of Language Modeling

```
lstmp = nn.SeqLSTMP(inputsize, hiddensize, outputsize)
```

The `SeqLSTMP` is a subclass of SeqLSTM.
It differs in that after computing the hidden state `h[t]` (eq. 6), it is
projected onto `r[t]` using a simple linear transform (eq. 7).
The computation of the gates also uses the previous such projection `r[t-1]` (eq. 1, 2, 3, 5).
This differs from `SeqLSTM` which uses `h[t-1]` instead of `r[t-1]`.

The computation of a time-step outlined in `SeqLSTM` is replaced with the following:

```
i[t] = σ(W[x->i]x[t] + W[r->i]r[t-1] + b[1->i])
(1)
f[t] = σ(W[x->f]x[t] + W[r->f]r[t-1] + b[1->f])
(2)
z[t] = tanh(W[x->c]x[t] + W[h->c]r[t-1] + b[1->c])
(3)
c[t] = f[t]c[t-1] + i[t]z[t]
(4)
o[t] = σ(W[x->o]x[t] + W[r->o]r[t-1] + b[1->o])
```

```
(5)
h[t] = o[t]tanh(c[t])
(6)
r[t] = W[h->r]h[t]
(7)
```

The algorithm is outlined in ref. A and benchmarked with state of the art results on the Google billion words dataset in ref. B.
`SeqLSTMP` can be used with an `hiddensize >> outputsize` such that the effective size of the memory cells `c[t]`
and gates `i[t]`, `f[t]` and `o[t]` can be much larger than the actual input `x[t]` and output `r[t]`.
For fixed `inputsize` and `outputsize`, the `SeqLSTMP` will be able to remember much more information than the `SeqLSTM`.

SeqGRU

This module is a faster version of `nn.Sequencer(nn.GRU(inputsize, outputsize))`:

```
seqGRU = nn.SeqGRU(inputsize, outputsize)
```

Usage of SeqGRU differs from GRU in the same manner as SeqLSTM differs from LSTM. Therefore see SeqLSTM for more details.

SeqBRNN

```
brnn = nn.SeqBRNN(inputSize, outputSize, [batchFirst], [merge])
```

A bi-directional RNN that uses SeqLSTM. Internally contains a 'fwd' and 'bwd' module of SeqLSTM. Expects an input shape of `seqlen x batchsize x inputsize`.
By setting [batchFirst] to true, the input shape can be `batchsize x seqLen x inputsize`.
Merge module defaults to CAddTable(), summing the outputs from each
output layer.

Example:

```
input = torch.rand(1, 1, 5)
brnn = nn.SeqBRNN(5, 5)
print(brnn:forward(input))
```

Prints an output of a 1x1x5 tensor.

BiSequencer

Applies encapsulated `fwd` and `bwd` rnns to an input sequence in forward and reverse order.
It is used for implementing Bidirectional RNNs and LSTMs.

```
brnn = nn.BiSequencer(fwd, [bwd, merge])
```

The input to the module is a sequence (a table) of tensors
and the output is a sequence (a table) of tensors of the same length.
Applies a `fwd` rnn (an AbstractRecurrent instance) to each element in the sequence in forward order and applies the `bwd` rnn in reverse order (from last element to first element).
The `bwd` rnn defaults to:

```
bwd = fwd:clone()
bwd:reset()
```

For each step (in the original sequence), the outputs of both rnns are merged together using the `merge` module (defaults to `nn.JoinTable(1,1)`).
If `merge` is a number, it specifies the JoinTable
constructor's `nInputDim` argument. Such that the `merge` module is then initialized as :

```
merge = nn.JoinTable(1,merge)
```

Internally, the `BiSequencer` is implemented by decorating a structure of modules that makes use of 3 Sequencers for the forward, backward and merge modules.

Similarly to a Sequencer, the sequences in a batch must have the same size.
But the sequence length of each batch can vary.

Note : make sure you call `brnn:forget()` after each call to `updateParameters()`.

Alternatively, one could call `brnn.bwdSeq:forget()` so that only `bwd` rnn forgets.
This is the minimum requirement, as it would not make sense for the `bwd` rnn to remember future sequences.

BiSequencerLM

Applies encapsulated `fwd` and `bwd` rnns to an input sequence in forward and reverse order.
It is used for implementing Bidirectional RNNs and LSTMs for Language Models (LM).

```
brnn = nn.BiSequencerLM(fwd, [bwd, merge])
```

The input to the module is a sequence (a table) of tensors
and the output is a sequence (a table) of tensors of the same length.
Applies a `fwd` rnn (an AbstractRecurrent instance to the
first `N-1` elements in the sequence in forward order.
Applies the `bwd` rnn in reverse order to the last `N-1` elements (from second-to-last element to first element).
This is the main difference of this module with the BiSequencer.
The latter cannot be used for language modeling because the `bwd` rnn would be trained to predict the input it had just be fed as input.

The `bwd` rnn defaults to:

```
bwd = fwd:clone()
bwd:reset()
```

While the `fwd` rnn will output representations for the last `N-1` steps,
the `bwd` rnn will output representations for the first `N-1` steps.
The missing outputs for each rnn (the first step for the `fwd` , the last step for the `bwd`)
will be filled with zero Tensors of the same size the commensure rnn's outputs.
This way they can be merged. If `nn.JoinTable` is used (the default), then the first
and last output elements will be padded with zeros for the missing `fwd` and `bwd` rnn outputs, respectively.

For each step (in the original sequence), the outputs of both rnns are merged together using
the `merge` module (defaults to `nn.JoinTable(1,1)`).
If `merge` is a number, it specifies the JoinTable

constructor's `nInputDim` argument. Such that the `merge` module is then initialized as :

```
merge = nn.JoinTable(1,merge)
```

Similarly to a Sequencer, the sequences in a batch must have the same size.
But the sequence length of each batch can vary.

Note that LMs implemented with this module will not be classical LMs as they won't measure the
probability of a word given the previous words. Instead, they measure the probabiliy of a word given the surrounding words, i.e. context. While for mathematical reasons you may not be able to use this to measure the
probability of a sequence of words (like a sentence),
you can still measure the pseudo-likeliness of such a sequence (see this for a discussion).

Repeater

This Module is a decorator similar to Sequencer.
It differs in that the sequence length is fixed before hand and the input is repeatedly forwarded through the wrapped `module` to produce an output table of length `nStep` :

```
r = nn.Repeater(module, nStep)
```

Argument `module` should be an `AbstractRecurrent` instance.
This is useful for implementing models like RCNNs,
which are repeatedly presented with the same input.

RecurrentAttention

References :

- A. Recurrent Models of Visual Attention
- B. Simple Statistical Gradient-Following Algorithms for Connectionist Reinforcement Learning

This module can be used to implement the Recurrent Attention Model (RAM) presented in Ref. A :

```
ram = nn.RecurrentAttention(rnn, action, nStep, hiddenSize)
```

`rnn` is an AbstractRecurrent instance.
Its input is `{x, z}` where `x` is the input to the `ram` and `z` is an action sampled from the `action` module.
The output size of the `rnn` must be equal to `hiddenSize`.

`action` is a Module
that uses a REINFORCE module (ref. B) like
ReinforceNormal,
ReinforceCategorical, or
ReinforceBernoulli
to sample actions given the previous time-step's output of the `rnn`.
During the first time-step, the `action` module is fed with a Tensor of zeros of size `input:size(1) x hiddenSize`.
It is important to understand that the sampled actions do not receive gradients backpropagated from the training criterion.
Instead, a reward is broadcast from a Reward Criterion like VRClassReward Criterion to the `action`'s REINFORCE module, which will backprogate graidents computed from the `output` samples
and the `reward`.
Therefore, the `action` module's outputs are only used internally, within the RecurrentAttention module.

`nStep` is the number of actions to sample, i.e. the number of elements in the `output` table.

`hiddenSize` is the output size of the `rnn`. This variable is necessary to generate the zero Tensor to sample an action for the first step (see above).

A complete implementation of Ref. A is available here.

MaskZero

This module zeroes the `output` rows of the decorated module for commensurate `input` rows which are tensors of zeros.

```
mz = nn.MaskZero(module, nInputDim)
```

The `output` Tensor (or table thereof) of the decorated `module`
will have each row (samples) zeroed when the commensurate row of the `input`
is a tensor of zeros.

The `nInputDim` argument must specify the number of non-batch dims
in the first Tensor of the `input`. In the case of an `input` table,
the first Tensor is the first one encountered when doing a depth-first search.

This decorator makes it possible to pad sequences with different lengths in the same batch with zero vectors.

Caveat: `MaskZero` not guarantee that the `output` and `gradInput` tensors of the internal modules
of the decorated `module` will be zeroed as well when the `input` is zero as well.
`MaskZero` only affects the immediate `gradInput` and `output` of the module that it encapsulates.
However, for most modules, the gradient update for that time-step will be zero because backpropagating a gradient of zeros will typically yield zeros all the way to the input.
In this respect, modules to avoid in encapsulating inside a `MaskZero` are `AbsractRecurrent`
instances as the flow of gradients between different time-steps internally.
Instead, call the AbstractRecurrent.maskZero method
to encapsulate the internal `recurrentModule`.

TrimZero

WARNING : only use this module if your input contains lots of zeros.
In almost all cases, `MaskZero` will be faster, especially with CUDA.

Ref. A : TrimZero: A Torch Recurrent Module for Efficient Natural Language Processing

The usage is the same with `MaskZero`.

```
mz = nn.TrimZero(module, nInputDim)
```

The only difference from `MaskZero` is that it reduces computational costs by varying a batch size, if any, for the case that varying lengths are provided in the input.

Notice that when the lengths are consistent, `MaskZero` will be faster, because `TrimZero` has an operational cost.

In short, the result is the same with `MaskZero`'s, however, `TrimZero` is faster than `MaskZero` only when sentence lengths is costly vary.

In practice, e.g. language model, `TrimZero` is expected to be faster than `MaskZero` about 30%. (You can test with it using `test/test_trimzero.lua`.)

LookupTableMaskZero

This module extends `nn.LookupTable` to support zero indexes. Zero indexes are forwarded as zero tensors.

```
lt = nn.LookupTableMaskZero(nIndex, nOutput)
```

The `output` Tensor will have each row zeroed when the commensurate row of the `input` is a zero index.

This lookup table makes it possible to pad sequences with different lengths in the same batch with zero vectors.

MaskZeroCriterion

This criterion zeroes the `err` and `gradInput` rows of the decorated criterion
for commensurate `input` rows which are tensors of zeros.

```
mzc = nn.MaskZeroCriterion(criterion, nInputDim)
```

The `gradInput` Tensor (or table thereof) of the decorated `criterion`
will have each row (samples) zeroed when the commensurate row of the `input`
is a tensor of zeros. The `err` will also disregard such zero rows.

The `nInputDim` argument must specify the number of non-batch dims
in the first Tensor of the `input`. In the case of an `input` table,
the first Tensor is the first one encountered when doing a depth-first search.

This decorator makes it possible to pad sequences with different lengths in the same batch with zero vectors.

SeqReverseSequence

```
reverseSeq = nn.SeqReverseSequence(dim)
```

Reverses an input tensor on a specified dimension. The reversal dimension can be no larger than three.

Example:

```
input = torch.Tensor({{1,2,3,4,5}, {6,7,8,9,10}})
reverseSeq = nn.SeqReverseSequence(1)
print(reverseSeq:forward(input))

Gives us an output of torch.Tensor({{6,7,8,9,10},{1,2,3,4,5}})
```

SequencerCriterion

This Criterion is a decorator:

```
c = nn.SequencerCriterion(criterion, [sizeAverage])
```

Both the `input` and `target` are expected to be a sequence, either as a table or Tensor.
For each step in the sequence, the corresponding elements of the input and target
will be applied to the `criterion`.
The output of `forward` is the sum of all individual losses in the sequence.
This is useful when used in conjunction with a Sequencer.

If `sizeAverage` is `true` (default is `false`), the `output` loss and `gradInput` is averaged over each time-step.

RepeaterCriterion

This Criterion is a decorator:

```
c = nn.RepeaterCriterion(criterion)
```

The `input` is expected to be a sequence (table or Tensor). A single `target` is repeatedly applied using the same `criterion` to each element in the `input` sequence. The output of `forward` is the sum of all individual losses in the sequence.
This is useful for implementing models like RCNNs,
which are repeatedly presented with the same target.

dpnn : deep extensions to nn

This package provides many useful features that aren't part of the main nn package.
These include sharedClone, which allows you to clone a module and share parameters or gradParameters with the original module, without incuring any memory overhead.
We also redefined type such that the type-cast preserves Tensor sharing within a structure of modules.

The package provides the following Modules:

- Decorator : abstract class to change the behaviour of an encapsulated module ;
- DontCast : prevent encapsulated module from being casted by `Module:type()` ;
- Serial : decorate a module makes its serialized output more compact ;
- NaN : decorate a module to detect the source of NaN errors ;
- Inception : implements the Inception module of the GoogleLeNet article ;
- Collapse : just like `nn.View(-1)` ;
- Convert : convert between different tensor types or shapes;
- ZipTable : zip a table of tables into a table of tables;
- ZipTableOneToMany : zip a table of element `el` and table of elements into a table of pairs of element `el` and table elements;
- CAddTensorTable : adds a tensor to a table of tensors of the same size;
- ReverseTable : reverse the order of elements in a table;
- PrintSize : prints the size of inputs and gradOutputs (useful for debugging);
- Clip : clips the inputs to a min and max value;
- Constant : outputs a constant value given an input (which is ignored);
- SpatialUniformCrop : uniformly crops patches from a input;
- SpatialGlimpse : takes a fovead glimpse of an image at a given location;
- WhiteNoise : adds isotropic Gaussian noise to the signal when in training mode;
- OneHot : transforms a tensor of indices into one-hot encoding;
- Kmeans : Kmeans clustering layer. Forward computes distances with respect to centroids and returns index of closest centroid. Centroids can be updated using gradient descent. Centroids could be initialized randomly or by using kmeans++ algoirthm;
- SpatialRegionDropout : Randomly dropouts a region (top, bottom, leftmost, rightmost) of the input image. Works with batch and any number of channels;
- FireModule : FireModule as mentioned in the SqueezeNet;
- NCEModule : optimized placeholder for a `Linear` + `SoftMax` using noise-contrastive estimation.
- SpatialFeatNormalization : Module for widely used preprocessing step of mean zeroing and standardization for images.
- SpatialBinaryConvolution : Module for binary spatial convolution (Binary weights) as

mentioned in XNOR-Net.
- SimpleColorTransform : Module for adding independent random noise to input image channels.
- PCAColorTransform : Module for adding noise to input image using Principal Components Analysis.

The following modules and criterions can be used to implement the REINFORCE algorithm :

- Reinforce : abstract class for REINFORCE modules;
- ReinforceBernoulli : samples from Bernoulli distribution;
- ReinforceNormal : samples from Normal distribution;
- ReinforceGamma : samples from Gamma distribution;
- ReinforceCategorical : samples from Categorical (Multinomial with one sample) distribution;
- VRClassReward : criterion for variance-reduced classification-based reward;
- BinaryClassReward : criterion for variance-reduced binary classification reward (like `VRClassReward`, but for binary classes);

Additional differentiable criterions
* BinaryLogisticRegression : criterion for binary logistic regression;
* SpatialBinaryLogisticRegression : criterion for pixel wise binary logistic regression;
* NCECriterion : criterion exclusively used with NCEModule.
* ModuleCriterion : adds an optional `inputModule` and `targetModule` before a decorated criterion;
* BinaryLogisticRegression : criterion for binary logistic regression.
* SpatialBinaryLogisticRegression : criterion for pixel wise binary logistic regression.

A lot of the functionality implemented here was pulled from
dp, which makes heavy use of this package.
However, dpnn can be used without dp (for e.g. you can use it with optim),
which is one of the main reasons why we made it.

Tutorials

Sagar Waghmare wrote a nice tutorial
on how to use dpnn with nngraph to reproduce the
Lateral Connections in Denoising Autoencoders Support Supervised Learning.

A brief (1 hours) overview of Torch7, which includes some details about **dpnn**,
is available via this NVIDIA GTC Webinar video. In any case, this presentation gives a nice
overview of Logistic Regression, Multi-Layer Perceptrons, Convolutional Neural Networks and

Recurrent Neural Networks using Torch7.

Module

The Module interface has been further extended with methods that facilitate stochastic gradient descent like updateGradParameters (i.e. momentum learning), weightDecay, maxParamNorm (for regularization), and so on.

Module.dpnn_parameters

A table that specifies the name of parameter attributes.
Defaults to `{'weight', 'bias'}`, which is a static variable (i.e. table exists in class namespace).
Sub-classes can define their own table statically.

Module.dpnn_gradParameters

A table that specifies the name of gradient w.r.t. parameter attributes.
Defaults to `{'gradWeight', 'gradBias'}`, which is a static variable (i.e. table exists in class namespace).
Sub-classes can define their own table statically.

[self] Module:type(type_str)

This function converts all the parameters of a module to the given `type_str`.
The `type_str` can be one of the types defined for torch.Tensor
like `torch.DoubleTensor`, `torch.FloatTensor` and `torch.CudaTensor`.
Unlike the type method
defined in nn, this one was overriden to
maintain the sharing of storage
among Tensors. This is especially useful when cloning modules share `parameters` and `gradParameters`.

[clone] Module:sharedClone([shareParams, shareGradParams])

Similar to clone.
Yet when `shareParams = true` (the default), the cloned module will share the parameters with the original module.
Furthermore, when `shareGradParams = true` (the default), the clone module will share the gradients w.r.t. parameters with the original module.
This is equivalent to :

```
clone = mlp:clone()
clone:share(mlp, 'weight', 'bias', 'gradWeight', 'gradBias')
```

yet it is much more efficient, especially for modules with lots of parameters, as these Tensors aren't needlessly copied during the `clone`.
This is particularly useful for Recurrent neural networks
which require efficient copies with shared parameters and gradient w.r.t. parameters for each time-step.

Module:maxParamNorm([maxOutNorm, maxInNorm])

This method implements a hard constraint on the upper bound of the norm of output and/or input neuron weights
(Hinton et al. 2012, p. 2) .
In a weight matrix, this is a contraint on rows (`maxOutNorm`) and/or columns (`maxInNorm`), respectively.
Has a regularization effect analogous to weightDecay, but with easier to optimize hyper-parameters.
Assumes that parameters are arranged (`output dim x ... x input dim`).
Only affects parameters with more than one dimension.
The method should normally be called after updateParameters.
It uses the C/CUDA optimized torch.renorm function.
Hint : `maxOutNorm = 2` usually does the trick.

[momGradParams] Module:momentumGradParameters()

Returns a table of Tensors (`momGradParams`). For each element in the table, a corresponding parameter (`params`) and gradient w.r.t. parameters (`gradParams`) is returned by a call to parameters.
This method is used internally by updateGradParameters.

Module:updateGradParameters(momFactor [, momDamp, momNesterov])

Applies classic momentum or Nesterov momentum (Sutskever, Martens et al, 2013) to parameter gradients.
Each parameter Tensor (`params`) has a corresponding Tensor of the same size for gradients w.r.t. parameters (`gradParams`).
When using momentum learning, another Tensor is added for each parameter Tensor (`momGradParams`).
This method should be called before updateParameters
as it affects the gradients w.r.t. parameters.

Classic momentum is computed as follows :

```
momGradParams = momFactor*momGradParams + (1-momDamp)*gradParams
gradParams = momGradParams
```

where `momDamp` has a default value of `momFactor` .

Nesterov momentum (`momNesterov = true`) is computed as follows (the first line is the same as classic momentum):

```
momGradParams = momFactor*momGradParams + (1-momDamp)*gradParams
gradParams = gradParams + momFactor*momGradParams
```

The default is to use classic momentum (`momNesterov = false`).

Module:weightDecay(wdFactor [, wdMinDim])

Decays the weight of the parameterized models.
Implements an L2 norm loss on parameters with dimensions greater or equal to `wdMinDim` (default is 2).

The resulting gradients are stored into the corresponding gradients w.r.t. parameters.
Such that this method should be called before updateParameters.

Module:gradParamClip(cutoffNorm [, moduleLocal])

Implements a contrainst on the norm of gradients w.r.t. parameters (Pascanu et al. 2012).
When `moduleLocal = false` (the default), the norm is calculated globally to Module for which this is called.
So if you call it on an MLP, the norm is computed on the concatenation of all parameter Tensors.
When `moduleLocal = true`, the norm constraint is applied
to the norm of all parameters in each component (non-container) module.
This method is useful to prevent the exploding gradient in
Recurrent neural networks.

Module:reinforce(reward)

This method is used by Criterions that implement the REINFORCE algorithm like VRClassReward.
While vanilla backpropagation (gradient descent using the chain rule),
REINFORCE Criterions broadcast a `reward` to all REINFORCE modules between the `forward` and the `backward`.
In this way, when the following call to `backward` reaches the REINFORCE modules,
these will compute a `gradInput` using the broadcasted `reward`.
The `reward` is broadcast to all REINFORCE modules contained
within `model` by calling `model:reinforce(reward)`.
Note that the `reward` should be a 1D tensor of size `batchSize`,
i.e. each example in a batch has its own scalar reward.

Refer to this example
for a complete training script making use of the REINFORCE interface.

Decorator

```
dmodule = nn.Decorator(module)
```

This module is an abstract class used to decorate a `module`. This means that method calls to `dmodule` will call the same method on the encapsulated `module`, and return its results.

DontCast

```
dmodule = nn.DontCast(module)
```

This module is a decorator. Use it to decorate a module that you don't want to be cast when the `type()` method is called.

```
module = nn.DontCast(nn.Linear(3,4):float())
module:double()
th> print(module:forward(torch.FloatTensor{1,2,3}))
 1.0927
-1.9380
-1.8158
-0.0805
[torch.FloatTensor of size 4]
```

Serial

```
dmodule = nn.Serial(module, [tensortype])
dmodule:[light,medium,heavy]Serial()
```

This module is a decorator that can be used to control the serialization/deserialization behavior of the encapsulated module. Basically, making the resulting string or file heavy (the default), medium or light in terms of size.

Furthermore, when specified, the `tensortype` attribute (e.g *torch.FloatTensor*, *torch.DoubleTensor* and so on.),
determines what type the module will be cast to during serialization.
Note that this will also be the type of the deserialized object.

The default serialization `tensortype` is `nil`, i.e. the module is serialized as is.

The `heavySerial()` has the serialization process serialize every attribute in the module graph,
which is the default behavior of nn.

The `mediumSerial()` has the serialization process serialize
everything except the attributes specified in each module's `dpnn_mediumEmpty`
table, which has a default value of `{'output', 'gradInput', 'momGradParams', 'dpnn_input'}`.
During serialization, whether they be tables or Tensors, these attributes are emptied (no storage).
Some modules overwrite the default `Module.dpnn_mediumEmpty` static attribute with their own.

The `lightSerial()` has the serialization process empty
everything a call to `mediumSerial(type)` would (so it uses `dpnn_mediumEmpty`).
But also empties all the parameter gradients specified by the
attribute `dpnn_gradParameters`, which defaults to `{gradWeight, gradBias}`.

We recomment using `mediumSerial()` for training, and `lightSerial()` for production (feed-forward-only models).

NaN

```
dmodule = nn.NaN(module, [id])
```

The `NaN` module asserts that the `output` and `gradInput` of the decorated `module` do not contain NaNs.
This is useful for locating the source of those pesky NaN errors.
The `id` defaults to automatically incremented values of `1,2,3,...`.

For example:

```
linear = nn.Linear(3,4)
mlp = nn.Sequential()
mlp:add(nn.NaN(nn.Identity()))
mlp:add(nn.NaN(linear))
mlp:add(nn.NaN(nn.Linear(4,2)))
print(mlp)
```

As you can see the `NaN` layers are have unique ids :

```
nn.Sequential {
  [input -> (1) -> (2) -> (3) -> output]
  (1): nn.NaN(1) @ nn.Identity
  (2): nn.NaN(2) @ nn.Linear(3 -> 4)
  (3): nn.NaN(3) @ nn.Linear(4 -> 2)
}
```

And if we fill the `bias` of the linear module with NaNs and call `forward` :

```
nan = math.log(math.log(0)) -- this is a nan value
linear.bias:fill(nan)
mlp:forward(torch.randn(2,3))
```

We get a nice error message:

```
/usr/local/share/lua/5.1/dpnn/NaN.lua:39: NaN found in parameters
of module :
nn.NaN(2) @ nn.Linear(3 -> 4)
```

Inception

References :

- A. Going Deeper with Convolutions
- B. GoogleLeNet

```
module = nn.Inception(config)
```

This module uses `n` +2 parallel “columns”.
The original paper uses 2+2 where the first two are (but there could be more than two):

- 1x1 conv (reduce) -> relu -> 5x5 conv -> relu
- 1x1 conv (reduce) -> relu -> 3x3 conv -> relu

and where the other two are :

- 3x3 maxpool -> 1x1 conv (reduce/project) -> relu
- 1x1 conv (reduce) -> relu.

This module allows the first group of columns to be of any
number while the last group consist of exactly two columns.
The 1x1 convoluations are used to reduce the number of input channels
(or filters) such that the capacity of the network doesn't explode.
We refer to these here has *reduce*.
Since each column seems to have one and only one reduce, their initial
configuration options are specified in lists of n+2 elements.

The sole argument `config` is a table taking the following key-values :

- Required Arguments :
  - `inputSize` : number of input channels or colors, e.g. 3;
  - `outputSize` : numbers of filters in the non-1x1 convolution kernel sizes, e.g. `{32,48}`
  - `reduceSize` : numbers of filters in the 1x1 convolutions (reduction) used in each column, e.g. `{48,64,32,32}` . The last 2 are used respectively for the max pooling (projection) column (the last column in the paper) and the column that has nothing but a 1x1 conv (the first column in the paper). This table should have two elements more than the outputSize

- Optional Arguments :
  - `reduceStride` : strides of the 1x1 (reduction) convolutions. Defaults to `{1,1,...}` .
  - `transfer` : transfer function like `nn.Tanh` , `nn.Sigmoid` , `nn.ReLU` , `nn.Identity` , etc. It is used after each reduction (1x1 convolution) and convolution. Defaults to `nn.ReLU` .
  - `batchNorm` : set this to `true` to use batch normalization. Defaults to `false` . Note that batch normalization can be awesome
  - `padding` : set this to `true` to add padding to the input of the convolutions such that output width and height are same as that of the original non-padded `input` . Defaults to `true` .
  - `kernelSize` : size (`height = width`) of the non-1x1 convolution kernels. Defaults to `{5,3}` .
  - `kernelStride` : stride of the kernels (`height = width`) of the convolution. Defaults to `{1,1}`
  - `poolSize` : size (`height = width`) of the spatial max pooling used in the next-to-last column. Defaults to 3.
  - `poolStride` : stride (`height = width`) of the spatial max pooling. Defaults to 1.

For a complete example using this module, refer to the following :
* deep inception training script ;
* openface facial recognition (the model definition is here).

Collapse

```
module = nn.Collapse(nInputDim)
```

This module is the equivalent of:

```
view = nn.View(-1)
view:setNumInputDim(nInputDim)
```

It collapses all non-batch dimensions. This is useful for converting
a spatial feature map to the single dimension required by a dense
hidden layer like Linear.

Convert

```
module = nn.Convert([inputShape, outputShape])
```

Module to convert between different data formats.
For example, we can flatten images by using :

```
module = nn.Convert('bchw', 'bf')
```

or equivalently

```
module = nn.Convert('chw', 'f')
```

Lets try it with an input:

```
print(module:forward(torch.randn(3,2,3,1)))
 0.5692 -0.0190  0.5243  0.7530  0.4230  1.2483
-0.9142  0.6013  0.5608 -1.0417 -1.4014  1.0177
-1.5207 -0.1641 -0.4166  1.4810 -1.1725 -1.0037
[torch.DoubleTensor of size 3x6]
```

You could also try:

```
module = nn.Convert('chw', 'hwc')
input = torch.randn(1,2,3,2)
input:select(2,1):fill(1)
input:select(2,2):fill(2)
print(input)
(1,1,.,.) =
  1  1
  1  1
  1  1
(1,2,.,.) =
  2  2
  2  2
  2  2
[torch.DoubleTensor of size 1x2x3x2]
print(module:forward(input))
(1,1,.,.) =
  1  2
  1  2

(1,2,.,.) =
  1  2
  1  2

(1,3,.,.) =
  1  2
  1  2
[torch.DoubleTensor of size 1x3x2x2]
```

Furthermore, it automatically converts the `input` to have the same type as `self.output` (i.e. the type of the module).
So you can also just use is for automatic input type converions:

```
module = nn.Convert()
print(module.output) -- type of module
```

```
[torch.DoubleTensor with no dimension]
input = torch.FloatTensor{1,2,3}
print(module:forward(input))
 1
 2
 3
[torch.DoubleTensor of size 3]
```

ZipTable

```
module = nn.ZipTable()
```

Zips a table of tables into a table of tables.

Example:

```
print(module:forward{ {'a1','a2'}, {'b1','b2'}, {'c1','c2'} })
{ {'a1','b1','c1'}, {'a2','b2','c2'} }
```

ZipTableOneToMany

```
module = nn.ZipTableOneToMany()
```

Zips a table of element `el` and table of elements `tab` into a table of tables, where the i-th table contains the element `el` and the i-th element in table `tab`

Example:

```
print(module:forward{ 'el', {'a','b','c'} })
{ {'el','a'}, {'el','b'}, {'el','c'} }
```

CAddTensorTable

```
module = nn.CAddTensorTable()
```

Adds the first element `el` of the input table `tab` to each tensor contained in the second element of `tab`, which is itself a table

Example:

```
print(module:forward{ (0,1,1), {(0,0,0),(1,1,1)} })
{ (0,1,1), (1,2,2) }
```

ReverseTable

```
module = nn.ReverseTable()
```

Reverses the order of elements in a table.

Example:

```
print(module:forward{1,2,3,4})
{4,3,2,1}
```

PrintSize

```
module = nn.PrintSize(name)
```

This module is useful for debugging complicated module composites.
It prints the size of the `input` and `gradOutput` during `forward`

and `backward` propagation respectively.
The `name` is a string used to identify the module along side the printed size.

Clip

```
module = nn.Clip(minval, maxval)
```

This module clips `input` values such that the output is between `minval` and `maxval`.

Constant

```
module = nn.Constant(value, nInputDim)
```

This module outputs a constant value given an input.
If `nInputDim` is specified, it uses the input to determine the size of the batch.
The `value` is then replicated over the batch.
Otherwise, the `value` Tensor is output as is.
During `backward`, the returned `gradInput` is a zero Tensor of the same size as the `input`.
This module has no trainable parameters.

You can use this with nn.ConcatTable() to append constant inputs to an input :

```
nn.ConcatTable():add(nn.Constant(v)):add(nn.Identity())
```

This is useful when you want to output a value that is independent of the
input to the neural network (see this example).

SpatialUniformCrop

```
module = nn.SpatialUniformCrop(oheight, owidth)
```

During training, this module will output a cropped patch of size `oheight, owidth`
within the boundaries of the `input` image.
For each example, a location is sampled from a uniform distribution
such that each possible patch has an equal probability of being sampled.

During evaluation, the center patch is cropped and output.

This module is commonly used at the input layer to artificially
augment the size of the dataset to prevent overfitting.

SpatialGlimpse

Ref. A. Recurrent Model for Visual Attention

```
module = nn.SpatialGlimpse(size, depth, scale)
```

A glimpse is the concatenation of down-scaled cropped images of
increasing scale around a given location in a given image.
The input is a pair of Tensors: `{image, location}`
`location` are `(y,x)` coordinates of the center of the different scales
of patches to be cropped from image `image`.
Coordinates are between `(-1,-1)` (top-left) and `(1,1)` (bottom-right).
The `output` is a batch of glimpses taken in image at location `(y,x)`.

`size` can be either a scalar which specifies the `width = height` of glimpses,
or a table of `{height, width}` to support a rectangular shape of glimpses.
`depth` is number of patches to crop per glimpse (one patch per depth).
`scale` determines the `size(t) = scale * size(t-1)` of successive cropped patches.

So basically, this module can be used to focus the attention of the model
on a region of the input `image`.
It is commonly used with the RecurrentAttention
module (see this example).

WhiteNoise

```
module = nn.WhiteNoise([mean, stdev])
```

Useful in training [Denoising Autoencoders] (http://arxiv.org/pdf/1507.02672v1.pdf).
Takes `mean` and `stdev` of the normal distribution as input.
Default values for mean and standard deviation are 0 and 0.1 respectively.
With `module:training()`, noise is added during forward.
During `backward` gradients are passed as it is.
With `module:evaluate()` the mean is added to the input.

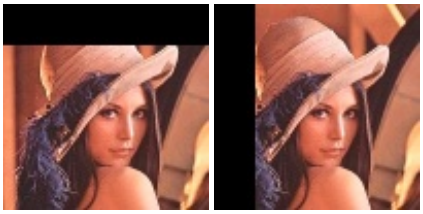
SpatialRegionDropout

```
module = nn.SpatialRegionDropout(p)
```

Following is an example of `SpatialRegionDropout` outputs on the famous lena image.

Input

Outputs



FireModule

Ref: http://arxiv.org/pdf/1602.07360v1.pdf

```
module = nn.FireModule(nInputPlane, s1x1, e1x1, e3x3, activation)
```

FireModule is comprised of two submodules 1) A *squeeze* convolution module comprised of `1x1` filters followed by 2) an *expand* module that is comprised of a mix of `1x1` and `3x3` convolution filters.
Arguments: `s1x1` : number of `1x1` filters in the squeeze submodule, `e1x1` : number of `1x1` filters in the expand submodule, `e3x3` : number of `3x3` filters in the expand submodule. It is recommended that `s1x1` be less than `(e1x1+e3x3)` if you want to limit the number of input channels to the `3x3` filters in the expand submodule.
FireModule works only with batches, for single sample convert the sample to a batch of size 1.

SpatialFeatNormalization

```
module = nn.SpatialFeatNormalization(mean, std)
```

This module normalizies each feature channel of input image based on its corresponding mean and standard deviation scalar values. This module does not learn the `mean` and `std`, they are provided as arguments.

SpatialBinaryConvolution

```
module = nn.SpatialBinaryConvolution(nInputPlane, nOutputPlane, kW,
kH)
```

Functioning of SpatialBinaryConvolution is similar to nn/SpatialConvolution. Only difference is that Binary weights are used for forward/backward and floating point weights are used for weight updates. Check **Binary-Weight-Network** section of XNOR-net.

SimpleColorTransform

```
range = torch.rand(inputChannels) -- Typically range is specified
by user.
module = nn.SimpleColorTransform(inputChannels, range)
```

This module performs a simple data augmentation technique. SimpleColorTransform module adds random noise to each color channel independently. In more advanced data augmentation technique noise is added using principal components of color channels. For that please check **PCAColorTransform**

PCAColorTransform

```
eigenVectors = torch.rand(inputChannels, inputChannels) -- Eigen
Vectors
eigenValues = torch.rand(inputChannels) -- Eigen
std = 0.1 -- Std deviation of normal distribution with mean zero
for noise.
module = nn.PCAColorTransform(inputChannels, eigenVectors,
eigenValues, std)
```

This module performs a data augmentation using Principal Component analysis of pixel values. When in training mode, mulitples of principal components are added to input image pixels. Magnitude of value added (noise) is dependent upon the corresponding eigen value and a random value sampled from a Gaussian distribution with mean zero and `std` (default 0.1) standard deviation. This technique was used in the famous AlexNet paper.

OneHot

```
module = nn.OneHot(outputSize)
```

Transforms a tensor of `input` indices having integer values between 1 and `outputSize` into a tensor of one-hot vectors of size `outputSize`.

Forward an index to get a one-hot vector :

```
> module = nn.OneHot(5) -- 5 classes
> module:forward(torch.LongTensor{3})
 0  0  1  0  0
[torch.DoubleTensor of size 1x5]
```

Forward a batch of 3 indices. Notice that these need not be stored as `torch.LongTensor` :

```
> module:forward(torch.Tensor{3,2,1})
 0  0  1  0  0
 0  1  0  0  0
 1  0  0  0  0
[torch.DoubleTensor of size 3x5]
```

Forward batch of `2 x 3` indices :

```
oh:forward(torch.Tensor{{3,2,1},{1,2,3}})
(1,.,.) =
  0  0  1  0  0
  0  1  0  0  0
  1  0  0  0  0

(2,.,.) =
  1  0  0  0  0
  0  1  0  0  0
  0  0  1  0  0
[torch.DoubleTensor of size 2x3x5]
```

Kmeans

```
km = nn.Kmeans(k, dim)
```

`k` is the number of centroids and `dim` is the dimensionality of samples.
You can either initialize centroids randomly from input samples or by using *kmeans++* algorithm.

```
km:initRandom(samples) -- Randomly initialize centroids from input
```

```
samples.
km:initKmeansPlus(samples) -- Use Kmeans++ to initialize centroids.
```

Example showing how to use Kmeans module to do standard Kmeans clustering.

```
attempts = 10
iter = 100 -- Number of iterations
bestKm = nil
bestLoss = math.huge
learningRate = 1
for j=1, attempts do
   local km = nn.Kmeans(k, dim)
   km:initKmeansPlus(samples)
   for i=1, iter do
      km:zeroGradParameters()
      km:forward(samples) -- sets km.loss
      km:backward(samples, gradOutput) -- gradOutput is ignored

      -- Gradient Descent weight/centroids update
      km:updateParameters(learningRate)
   end

   if km.loss < bestLoss then
      bestLoss = km.loss
      bestKm = km:clone()
   end
end
```

`nn.Kmeans()` module maintains loss only for the latest forward. If you want to maintain loss over the whole dataset then you who would need do it my adding the module loss for every forward.

You can also use `nn.Kmeans()` as an auxillary layer in your network.
A call to `forward` will generate an `output` containing the index of the nearest cluster for each sample in the batch.
The `gradInput` generated by `updateGradInput` will be zero.

ModuleCriterion

```
criterion = nn.ModuleCriterion(criterion [, inputModule,
targetModule, castTarget])
```

This criterion decorates a `criterion` by allowing the `input` and `target` to be fed through an optional `inputModule` and `targetModule` before being passed to the `criterion`. The `inputModule` must not contain parameters as these would not be updated.

When `castTarget = true` (the default), the `targetModule` is cast along with the `inputModule` and
`criterion`. Otherwise, the `targetModule` isn't.

NCEModule

Ref. A RNNLM training with NCE for Speech Recognition

```
ncem = nn.NCEModule(inputSize, outputSize, k, unigrams, [Z])
```

When used in conjunction with NCECriterion,
the `NCEModule` implements noise-contrastive estimation.

The point of the NCE is to speedup computation for large `Linear` + `SoftMax` layers.
Computing a forward/backward for `Linear(inputSize, outputSize)` for a large `outputSize` can be very expensive.
This is common when implementing language models having with large vocabularies of a million words.
In such cases, NCE can be an efficient alternative to computing the full `Linear` + `SoftMax` during training and
cross-validation.

The `inputSize` and `outputSize` are the same as for the `Linear` module.
The number of noise samples to be drawn per example is `k`. A value of 25 should work well.
Increasing it will yield better results, while a smaller value will be more efficient to process.
The `unigrams` is a tensor of size `outputSize` that contains the frequencies or probability distribution over classes.
It is used to sample noise samples via a fast implementation of `torch.multinomial`.
The `Z` is the normalization constant of the approximated SoftMax.
The default is `math.exp(9)` as specified in Ref. A.

For inference, or measuring perplexity, the full `Linear` + `SoftMax` will need to

be computed. The `NCEModule` can do this by switching on the following :

```
ncem:evaluate()
ncem.normalized = true
```

Furthermore, to simulate `Linear` + `LogSoftMax` instead, one need only add the following to the above:

```
ncem.logsoftmax = true
```

An example is provided via the rnn package.

NCECriterion

```
ncec = nn.NCECriterion()
```

This criterion only works with an NCEModule on the output layer.
Together, they implement noise-contrastive estimation.

Reinforce

Ref A. Simple Statistical Gradient-Following Algorithms for Connectionist Reinforcement Learning

Abstract class for modules that implement the REINFORCE algorithm (ref. A).

```
module = nn.Reinforce([stochastic])
```

The `reinforce(reward)` method is called by a special Reward Criterion (e.g. VRClassReward).
After which, when backward is called, the reward will be used to generate gradInputs.
When `stochastic=true`, the module is stochastic (i.e. samples from a distribution) during evaluation and training.

When `stochastic=false` (the default), the module is only stochastic during training.

The REINFORCE rule for a module can be summarized as follows :

```
              d ln(f(output,input))
gradInput = ---------------------  * reward
                    d input
```

where the `reward` is what is provided by a Reward criterion like
VRClassReward via the reinforce method.
The criterion will normally be responsible for the following formula :

```
reward = a*(R - b)
```

where `a` is the alpha of the original paper, i.e. a reward scale,
`R` is the raw reward (usually 0 or 1), and `b` is the baseline reward,
which is often taken to be the expected raw reward `R`.

The `output` is usually sampled from a probability distribution `f()`
parameterized by the `input`.
See ReinforceBernoulli for a concrete derivation.

Also, as you can see, the gradOutput is ignored. So within a backpropagation graph,
the `Reinforce` modules will replace the backpropagated gradients (`gradOutput`)
with their own obtained from the broadcasted `reward`.

ReinforceBernoulli

Ref A. Simple Statistical Gradient-Following Algorithms for
Connectionist Reinforcement Learning

```
module = nn.ReinforceBernoulli([stochastic])
```

A Reinforce subclass that implements the REINFORCE algorithm
(ref. A p.230-236) for the Bernoulli probability distribution.
Inputs are bernoulli probabilities `p`.
During training, outputs are samples drawn from this distribution.
During evaluation, when `stochastic=false`, outputs are the same as the inputs.

Uses the REINFORCE algorithm (ref. A p.230-236) which is
implemented through the reinforce interface (`gradOutputs` are ignored).

Given the following variables :

- `f` : bernoulli probability mass function
- `x` : the sampled values (0 or 1) (i.e. `self.output`)
- `p` : probability of sampling a 1

the derivative of the log bernoulli w.r.t. probability `p` is :

```
d ln(f(output,input))   d ln(f(x,p))    (x - p)
---------------------- = ------------ = ---------
      d input                d p         p(1 - p)
```

ReinforceNormal

Ref A. Simple Statistical Gradient-Following Algorithms for Connectionist Reinforcement Learning

```
module = nn.ReinforceNormal(stdev, [stochastic])
```

A Reinforce subclass that implements the REINFORCE algorithm
(ref. A p.238-239) for a Normal (i.e. Gaussian) probability distribution.
Inputs are the means of the normal distribution.
The `stdev` argument specifies the standard deviation of the distribution.
During training, outputs are samples drawn from this distribution.
During evaluation, when `stochastic=false` , outputs are the same as the inputs, i.e. the means.
Uses the REINFORCE algorithm (ref. A p.238-239) which is
implemented through the reinforce interface (`gradOutputs` are ignored).

Given the following variables :

- `f` : normal probability density function
- `x` : the sampled values (i.e. `self.output`)
- `u` : mean (`input`)
- `s` : standard deviation (`self.stdev`)

the derivative of log normal w.r.t. mean `u` is :

```
d ln(f(x,u,s))   (x - u)
-------------- = -------
     d u           s^2
```

As an example, it is used to sample locations for the RecurrentAttention
module (see this example).

ReinforceGamma

Ref A. Simple Statistical Gradient-Following Algorithms for Connectionist Reinforcement Learning

```
module = nn.ReinforceGamma(scale, [stochastic])
```

A Reinforce subclass that implements the REINFORCE algorithm
(ref. A) for a Gamma probability distribution
parametrized by shape (k) and scale (theta) variables.
Inputs are the shapes of the gamma distribution.
During training, outputs are samples drawn from this distribution.
During evaluation, when `stochastic=false`, outputs are equal to the mean, defined as the product of
shape and scale ie. `k*theta`.
Uses the REINFORCE algorithm (ref. A) which is
implemented through the reinforce interface (`gradOutputs` are ignored).

Given the following variables :

- `f` : gamma probability density function
- `g` : digamma function
- `x` : the sampled values (i.e. `self.output`)
- `k` : shape (`input`)
- `t` : scale

the derivative of log gamma w.r.t. shape `k` is :

```
d ln(f(x,k,t))
```

```
-------------- = ln(x) - g(k) - ln(t)
      d k
```

ReinforceCategorical

Ref A. Simple Statistical Gradient-Following Algorithms for Connectionist Reinforcement Learning

```
module = nn.ReinforceCategorical([stochastic])
```

A Reinforce subclass that implements the REINFORCE algorithm
(ref. A) for a Categorical (i.e. Multinomial with one sample) probability distribution.
Inputs are the categorical probabilities of the distribution : `p[1], p[2], ..., p[k]`.
These are usually the output of a SoftMax.
For `n` categories, both the `input` and `output` ares of size `batchSize x n`.
During training, outputs are samples drawn from this distribution.
The outputs are returned in one-hot encoding i.e.
the output for each example has exactly one category having a 1, while the remainder are zero.
During evaluation, when `stochastic=false`, outputs are the same as the inputs, i.e. the probabilities `p`.
Uses the REINFORCE algorithm (ref. A) which is
implemented through the reinforce interface (`gradOutputs` are ignored).

Given the following variables :

- `f` : categorical probability mass function
- `x` : the sampled indices (one per sample) (`self.output` is the one-hot encoding of these indices)
- `p` : probability vector (`p[1], p[2], ..., p[k]`) (`input`)

the derivative of log categorical w.r.t. probability vector `p` is :

```
d ln(f(x,p))     1/p[i]    if i = x
------------ =
    d p          0         otherwise
```

VRClassReward

Ref A. Simple Statistical Gradient-Following Algorithms for Connectionist Reinforcement Learning

This Reward criterion implements the REINFORCE algoritm (ref. A) for classification models. Specifically, it is a Variance Reduces (VR) classification reinforcement leanring (reward-based) criterion.

```
vcr = nn.VRClassReward(module [, scale, criterion])
```

While it conforms to the Criterion interface (which it inherits),
it does not backpropagate gradients (except for the baseline `b`; see below).
Instead, a `reward` is broadcast to the `module` via the reinforce method.

The criterion implements the following formula :

```
reward = a*(R - b)
```

where `a` is the alpha described in Ref. A, i.e. a reward `scale` (defaults to 1),
`R` is the raw reward (0 for incorrect and 1 for correct classification),
and `b` is the baseline reward, which is often taken to be the expected raw reward `R`.

The `target` of the criterion is a tensor of class indices.
The `input` to the criterion is a table `{y,b}` where `y` is the probability
(or log-probability) of classes (usually the output of a SoftMax),
and `b` is the baseline reward discussed above.

For each example, if `argmax(y)` is equal to the `target` class, the raw reward `R = 1`,
otherwize `R = 0`.

As for `b`, its `gradInputs` are obtained from the `criterion`, which defaults to `MSECriterion`.
The `criterion`'s target is the commensurate raw reward `R`.
Using `a*(R-b)` instead of `a*R` to obtain a `reward` is what makes this class variance reduced (VR).
By reducing the variance, the training can converge faster (Ref. A).
The predicted `b` can be nothing more than the expectation `E(R)`.

Note : for RNNs with R = 1 for last step in sequence, encapsulate it

in `nn.ModuleCriterion(VRClassReward, nn.SelectTable(-1))`.

For an example, this criterion is used along with the RecurrentAttention
module to train a recurrent model for visual attention.

BinaryClassReward

```
bcr = nn.BinaryClassReward(module [, scale, criterion])
```

This module implements VRClassReward for binary classification problems.
So basically, the `input` is still a table of two tensors.
The first input tensor is of size `batchsize` containing Bernoulli probabilities.
The second input tensor is the baseline prediction described in `VRClassReward`.
The targets contain zeros and ones.

BinaryLogisticRegression

Ref A. Learning to Segment Object Candidates
This criterion implements the score criterion mentioned in (ref. A).

```
criterion = nn.BinaryLogisticRegression()
```

BinaryLogisticRegression implements following cost function for binary classification.

```
log( 1 + exp( -y_k * score(x_k) ) )
```

where `y_k` is binary target `score(x_k)` is the corresponding prediction. `y_k` has value `{-1, +1}` and `score(x_k)` has value in `[-1, +1]`.

SpatialBinaryLogisticRegression

Ref A. Learning to Segment Object Candidates

This criterion implements the spatial component of the criterion mentioned in (ref. A).

```
criterion = nn.SpatialBinaryLogisticRegression()
```

SpatialBinaryLogisticRegression implements following cost function for binary pixel classification.

```
   1
_______ sum_ij [ log( 1 + exp( -m_ij * f_ij ) ) ]
 2*w*h
```

where `m_ij` is target binary image and `f_ij` is the corresponding prediction. `m_ij` has value `{-1, +1}` and `f_ij` has value in `[-1, +1]`.

Neural Network Graph Package

This package provides graphical computation for `nn` library in Torch.

Requirements

You do *not* need `graphviz` to be able to use this library, but if you have it you will be able to display the graphs that you have created. For installing the package run the appropriate command below:

```
# Mac users
brew install graphviz
# Debian/Ubuntu users
sudo apt-get install graphviz -y
```

Usage

Plug: A more explanatory nngraph tutorial by Nando De Freitas of Oxford

The aim of this library is to provide users of `nn` package with tools to easily create complicated architectures.
Any given `nn` `module` is going to be bundled into a *graph node*.
The `__call__` operator of an instance of `nn.Module` is used to create architectures as if one is writing function calls.

Two hidden layers MLP

```
h1 = nn.Linear(20, 10)()
h2 = nn.Linear(10, 1)(nn.Tanh()(nn.Linear(10, 10)(nn.Tanh()(h1))))
mlp = nn.gModule({h1}, {h2})
```

```
x = torch.rand(20)
dx = torch.rand(1)
mlp:updateOutput(x)
mlp:updateGradInput(x, dx)
mlp:accGradParameters(x, dx)

-- draw graph (the forward graph, '.fg')
graph.dot(mlp.fg, 'MLP')
```

MLP

Read this diagram from top to bottom, with the first and last nodes being *dummy nodes* that regroup all inputs and outputs of the graph.
The `module` entry describes the function of the node, as applies to `input`, and producing a result of the shape `gradOutput`; `mapindex` contains pointers to the parent nodes.

To save the *graph* on file, specify the file name, and both a `dot` and `svg` files will be saved. For example, you can type:

```
graph.dot(mlp.fg, 'MLP', 'myMLP')
```

You can also use the `__unm__` and `__sub__` operators to replace all `__call__`:

```
h1 = - nn.Linear(20,10)
h2 = h1
     - nn.Tanh()
     - nn.Linear(10,10)
     - nn.Tanh()
     - nn.Linear(10, 1)
mlp = nn.gModule({h1}, {h2})
```

A network with 2 inputs and 2 outputs

```
h1 = nn.Linear(20, 20)()
h2 = nn.Linear(10, 10)()
hh1 = nn.Linear(20, 1)(nn.Tanh()(h1))
hh2 = nn.Linear(10, 1)(nn.Tanh()(h2))
```

```
madd = nn.CAddTable()({hh1, hh2})
oA = nn.Sigmoid()(madd)
oB = nn.Tanh()(madd)
gmod = nn.gModule({h1, h2}, {oA, oB})

x1 = torch.rand(20)
x2 = torch.rand(10)

gmod:updateOutput({x1, x2})
gmod:updateGradInput({x1, x2}, {torch.rand(1), torch.rand(1)})
graph.dot(gmod.fg, 'Big MLP')
```

Alternatively, you can use `-` to make your code looks like the data flow:

```
h1 = - nn.Linear(20,20)
h2 = - nn.Linear(10,10)
hh1 = h1 - nn.Tanh() - nn.Linear(20,1)
hh2 = h2 - nn.Tanh() - nn.Linear(10,1)
madd = {hh1,hh2} - nn.CAddTable()
oA = madd - nn.Sigmoid()
oB = madd - nn.Tanh()
gmod = nn.gModule( {h1,h2}, {oA,oB} )
```

Big MLP

A network with containers

Another net that uses container modules (like `ParallelTable`) that output a table of outputs.

```
m = nn.Sequential()
m:add(nn.SplitTable(1))
m:add(nn.ParallelTable():add(nn.Linear(10, 20)):add(nn.Linear(10,
30)))
input = nn.Identity()()
input1, input2 = m(input):split(2)
m3 = nn.JoinTable(1)({input1, input2})

g = nn.gModule({input}, {m3})

indata = torch.rand(2, 10)
```

```
gdata = torch.rand(50)
g:forward(indata)
g:backward(indata, gdata)

graph.dot(g.fg, 'Forward Graph')
graph.dot(g.bg, 'Backward Graph')
```

Forward Graph

Backward Graph

More fun with graphs

A multi-layer network where each layer takes output of previous two layers as input.

```
input = nn.Identity()()
L1 = nn.Tanh()(nn.Linear(10, 20)(input))
L2 = nn.Tanh()(nn.Linear(30, 60)(nn.JoinTable(1)({input, L1})))
L3 = nn.Tanh()(nn.Linear(80, 160)(nn.JoinTable(1)({L1, L2})))

g = nn.gModule({input}, {L3})

indata = torch.rand(10)
gdata = torch.rand(160)
g:forward(indata)
g:backward(indata, gdata)

graph.dot(g.fg, 'Forward Graph')
graph.dot(g.bg, 'Backward Graph')
```

As your graph getting bigger and more complicated, the nested parentheses may become confusing. In this case, using `-` to chain the modules is a clearer and easier way:

```
input = - nn.Identity()
L1 =  input
     - nn.Linear(10, 20)
     - nn.Tanh()
L2 =  { input, L1 }
     -  nn.JoinTable(1)
     -  nn.Linear(30,60)
```

```
      -  nn.Tanh()
L3 = { L1,L2 }
      - nn.JoinTable(1)
      - nn.Linear(80,160)
      - nn.Tanh()
g = nn.gModule({input},{L3})
```

Forward Graph

Backward Graph

Annotations

It is possible to add annotations to your network, such as labeling nodes with names or attributes which will show up when you graph the network.
This can be helpful in large graphs.

For the full list of graph attributes see the
graphviz documentation.

```
input = nn.Identity()()
L1 = nn.Tanh()(nn.Linear(10, 20)(input)):annotate{
   name = 'L1', description = 'Level 1 Node',
   graphAttributes = {color = 'red'}
}
L2 = nn.Tanh()(nn.Linear(30, 60)(nn.JoinTable(1)({input,
L1}))):annotate{
   name = 'L2', description = 'Level 2 Node',
   graphAttributes = {color = 'blue', fontcolor = 'green'}
}
L3 = nn.Tanh()(nn.Linear(80, 160)(nn.JoinTable(1)({L1,
L2}))):annotate{
   name = 'L3', description = 'Level 3 Node',
   graphAttributes = {color = 'green',
   style = 'filled', fillcolor = 'yellow'}
}

g = nn.gModule({input},{L3})
```

```
indata = torch.rand(10)
gdata = torch.rand(160)
g:forward(indata)
g:backward(indata, gdata)

graph.dot(g.fg, 'Forward Graph', '/tmp/fg')
graph.dot(g.bg, 'Backward Graph', '/tmp/bg')
```

In this case, the graphs are saved in the following 4 files: `/tmp/{fg,bg}.{dot,svg}`.

Forward Graph

Backward Graph

Debugging

With nngraph, one can create very complicated networks. In these cases, finding errors can be hard. For that purpose, nngraph provides several useful utilities. The following code snippet shows how to use local variable names for annotating the nodes in a graph and how to enable debugging mode that automatically creates an svg file with error node marked in case of a runtime error.

```
require 'nngraph'

-- generate SVG of the graph with the problem node highlighted
-- and hover over the nodes in svg to see the filename:line_number
info
-- nodes will be annotated with local variable names even if debug
mode is not enabled.
nngraph.setDebug(true)

local function get_net(from, to)
    local from = from or 10
    local to = to or 10
    local input_x = nn.Identity()()
    local linear_module = nn.Linear(from, to)(input_x)

    -- Annotate nodes with local variable names
```

```
    nngraph.annotateNodes()
    return nn.gModule({input_x},{linear_module})
end

local net = get_net(10,10)

-- if you give a name to the net, it will use that name to produce
the
-- svg in case of error, if not, it will come up with a name
-- that is derived from number of inputs and outputs to the graph
net.name = 'my_bad_linear_net'

-- prepare an input that is of the wrong size to force an error
local input = torch.rand(11)
pcall(function() net:updateOutput(input) end)
-- it should have produced an error and spit out a graph
-- just run Safari to display the svg
os.execute('open -a  Safari my_bad_linear_net.svg')
```

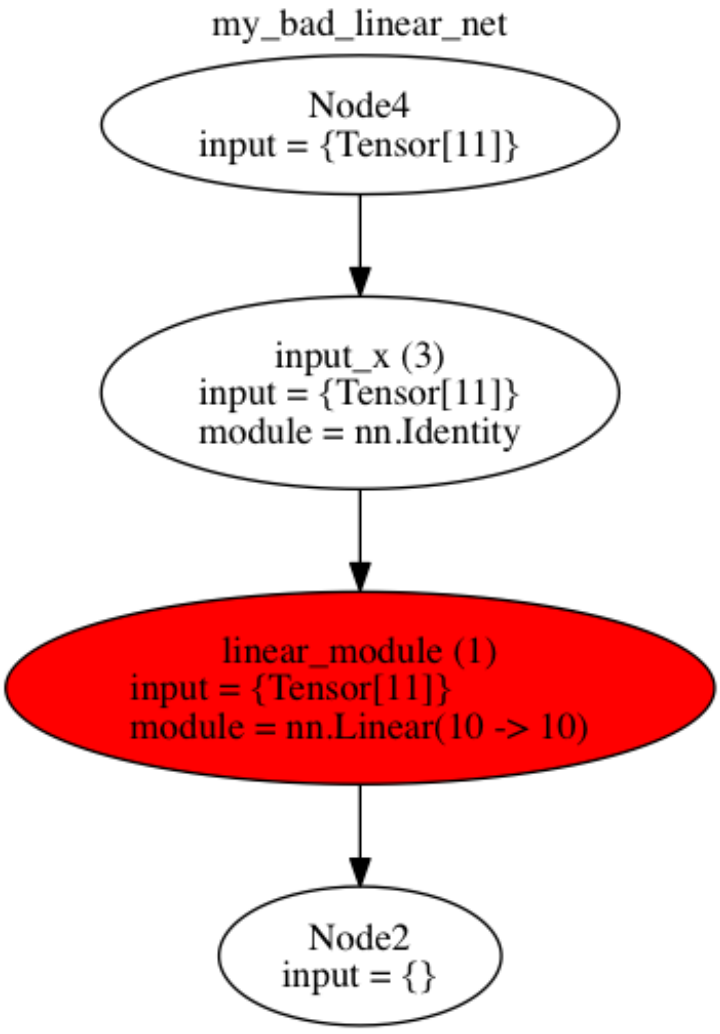


image Package Reference Manual

image is the Torch7 distribution package for processing
images. It contains a wide variety of functions divided into the following categories:

- Saving and loading images as JPEG, PNG, PPM and PGM;
- Simple transformations like translation, scaling and rotation;
- Parameterized transformations like convolutions and warping;
- Simple Drawing Routines like drawing text or a rectangle on an image;
- Graphical user interfaces like display and window;
- Color Space Conversions from and to RGB, YUV, Lab, and HSL;
- Tensor Constructors for creating Lenna, Fabio and Gaussian and Laplacian kernels;

Note that unless speficied otherwise, this package deals with images of size
`nChannel x height x width`.

Install

The easiest way to install this package it by following the intructions
to install Torch7, which includes **image**.
Otherwise, to update or manually re-install it:

```
$ luarocks install image
```

You can test your install with:

```
$ luajit -limage -e "image.test()"
```

Usage

```
> require 'image'
> l = image.lena()
> image.display(l)
> f = image.fabio()
> image.display(f)
```

Saving and Loading

This sections includes functions for saving and loading different types
of images to and from disk.

[res] image.load(filename, [depth, tensortype])

Loads an image located at path `filename` having `depth` channels (1 or 3)
into a Tensor
of type `tensortype` (*float*, *double* or *byte*). The last two arguments
are optional.

The image format is determined from the `filename`'s
extension suffix. Supported formats are
JPEG,
PNG,
PPM and PGM.

The returned `res` Tensor has size `nChannel x height x width` where `nChannel` is
1 (greyscale) or 3 (usually RGB
or YUV.

Usage:

```
--To load as byte tensor for rgb imagefile
local img = image.load(imagefile,3,'byte')

--To load as byte tensor for gray imagefile
local img = image.load(imagefile,1,'byte')
```

[res] image.getSize(filename)

Return the size of an image located at path `filename` into a LongTensor.

The image format is determined from the `filename`'s
extension suffix. Supported formats are
JPEG,

PNG,
PPM and PGM.

The returned `res` Tensor has size `3` (nChannel, height, width).

image.save(filename, tensor)

Saves Tensor `tensor` to disk at path `filename`. The format to which
the image is saved is extrapolated from the `filename`'s extension suffix.
The `tensor` should be of size `nChannel x height x width`.
To save with a minimal loss, the tensor values should lie in the range [0, 1] since the tensor is clamped between 0 and 1 before being saved to the disk.

[res] image.decompressJPG(tensor, [depth, tensortype])

Decompresses an image from a ByteTensor in memory having `depth` channels (1 or 3)
into a Tensor
of type `tensortype` (*float*, *double* or *byte*). The last two arguments
are optional.

Usage:

```
local fin = torch.DiskFile(imfile, 'r')
fin:binary()
fin:seekEnd()
local file_size_bytes = fin:position() - 1
fin:seek(1)
local img_binary = torch.ByteTensor(file_size_bytes)
fin:readByte(img_binary:storage())
fin:close()
-- Then when you're ready to decompress the ByteTensor:
im = image.decompressJPG(img_binary)
```

[res] image.compressJPG(tensor, [quality])

Compresses an image to a ByteTensor in memory. Optional quality is between 1 and 100 and adjusts compression quality.

Tensor Constructors

The following functions construct Tensors like Gaussian or
Laplacian kernels, or images like Lenna and Fabio.

[res] image.lena()

Returns the classic `Lenna.jpg` image as a `3 x 512 x 512` Tensor.

[res] image.fabio()

Returns the `fabio.jpg` image as a `257 x 271` Tensor.

[res] image.gaussian([size, sigma, amplitude, normalize, […]])

Returns a 2D Gaussian
kernel of size `height x width`. When used as a Gaussian smoothing operator in a 2D
convolution, this kernel is used to `blur` images and remove detail and noise
(ref.: Gaussian Smoothing).
Optional arguments `[...]` expand to
`width`, `height`, `sigma_horz`, `sigma_vert`, `mean_horz`, `mean_vert` and `tensor`.

The default value of `height` and `width` is `size`, where the latter
has a default value of 3. The amplitude of the Gaussian (its maximum value)
is `amplitude`. The default is 1.
When `normalize=true`, the kernel is normalized to have a sum of 1.
This overrides the `amplitude` argument. The default is `false`.
The default value of the horizontal and vertical standard deviation
`sigma_horz` and `sigma_vert` of the Gaussian kernel is `sigma`, where
the latter has a default value of 0.25. The default values for the
corresponding means `mean_horz` and `mean_vert` are 0.5. Both the
standard deviations and means are relative to kernels of unit width and height
where the top-left corner is the origin. In other works, a mean of 0.5 is
the center of the kernel size, while a standard deviation of 0.25 is a quarter

of it. When `tensor` is provided (a 2D Tensor), the `height`, `width` and `size` are ignored.
It is used to store the returned gaussian kernel.

Note that arguments can also be specified as key-value arguments (in a table).

[res] image.gaussian1D([size, sigma, amplitude, normalize, mean, tensor])

Returns a 1D Gaussian kernel of size `size`, mean `mean` and standard deviation `sigma`.
Respectively, these arguments have default values of 3, 0.25 and 0.5.
The amplitude of the Gaussian (its maximum value) is `amplitude`. The default is 1.
When `normalize=true`, the kernel is normalized to have a sum of 1.
This overrides the `amplitude` argument. The default is `false`. Both the standard deviation and mean are relative to a kernel of unit size.
In other works, a mean of 0.5 is the center of the kernel size, while a standard deviation of 0.25 is a quarter of it.
When `tensor` is provided (a 1D Tensor), the `size` is ignored.
It is used to store the returned gaussian kernel.

Note that arguments can also be specified as key-value arguments (in a table).

[res] image.laplacian([size, sigma, amplitude, normalize, [...]])

Returns a 2D Laplacian kernel of size `height x width`.
When used in a 2D convolution, the Laplacian of an image highlights regions of rapid intensity change and is therefore often used for edge detection (ref.: Laplacian/Laplacian of Gaussian).
Optional arguments `[...]` expand to `width`, `height`, `sigma_horz`, `sigma_vert`, `mean_horz`, `mean_vert`.

The default value of `height` and `width` is `size`, where the latter has a default value of 3. The amplitude of the Laplacian (its maximum value) is `amplitude`. The default is 1.
When `normalize=true`, the kernel is normalized to have a sum of 1.
This overrides the `amplitude` argument. The default is `false`.

The default value of the horizontal and vertical standard deviation `sigma_horz` and `sigma_vert` of the Laplacian kernel is `sigma`, where the latter has a default value of 0.25. The default values for the corresponding means `mean_horz` and `mean_vert` are 0.5. Both the standard deviations and means are relative to kernels of unit width and height where the top-left corner is the origin. In other works, a mean of 0.5 is the center of the kernel size, while a standard deviation of 0.25 is a quarter of it.

[res] image.colormap(nColor)

Creates an optimally-spaced RGB color mapping of `nColor` colors.
Note that the mapping is obtained by generating the colors around the HSV wheel, varying the Hue component.
The returned `res` Tensor has size `nColor x 3`.

[res] image.jetColormap(nColor)

Creates a jet (blue to red) RGB color mapping of `nColor` colors.
The returned `res` Tensor has size `nColor x 3`.

Simple Transformations

This section includes simple but very common image transformations
like cropping, translation, scaling and rotation.

[res] image.crop([dst,] src, x1, y1, [x2, y2])

Crops image `src` at coordinate `(x1, y1)` up to coordinate
`(x2, y2)`. The coordinate indexing is zero-based and `(x2, y2)` is non-inclusive.
If `dst` is provided, it is used to store the output
image. Otherwise, returns a new `res` Tensor.

```
-- The indexing starts with 0 and 2 is non-inclusive coordinate.
> require('image')
> image.crop(torch.Tensor(3, 2, 2), 0, 0 , 2, 2) -- crop is a
correct crop and the result is 3x2x2 tensor.
(1,.,.) =
  0  0
  0  0

(2,.,.) =
  0  0
  0  0

(3,.,.) =
  0  0
  0  0
[torch.DoubleTensor of size 3x2x2]
```

[res] image.crop([dst,] src, format, width, height)

Crops a `width x height` section of source image `src`. The argument
`format` is a string specifying where to crop: it can be “c”, “tl”, “tr”,
“bl” or “br” for center, top left, top right, bottom left and bottom right,
respectively. If `dst` is provided, it is used to store the output
image. Otherwise, returns a new `res` Tensor.

[res] image.translate([dst,] src, x, y)

Translates image `src` by `x` pixels horizontally and `y` pixels vertically. If `dst` is provided, it is used to store the output image. Otherwise, returns a new `res` Tensor.

[res] image.scale(src, width, height, [mode])

Rescale the height and width of image `src` to have width `width` and height `height`. Variable `mode` specifies type of interpolation to be used. Valid values include bilinear (the default), bicubic, or *simple* interpolation. Returns a new `res` Tensor.

[res] image.scale(src, size, [mode])

Rescale the height and width of image `src`. Variable `size` is a number or a string specifying the size of the result image. When `size` is a number, it specifies the maximum height or width of the output. When it is a string like `WxH` or `MAX` or `^MIN`, `*SC` or `*SCn/SCd` it specifies the `height x width`, maximum height or width of the output, minimum height or width of the output, scaling factor (number), or fractional scaling factor (int/int), respectively.

[res] image.scale(dst, src, [mode])

Rescale the height and width of image `src` to fit the dimensions of Tensor `dst`.

[res] image.rotate([dst,], src, theta, [mode])

Rotates image `src` by `theta` radians.
If `dst` is specified it is used to store the results of the rotation.

Variable `mode` specifies type of interpolation to be used. Valid values include *simple* (the default) or *bilinear* interpolation.

[res] image.polar([dst,], src, [interpolation], [mode])

Converts image `src` to polar coordinates. In the polar image, angular information is in the vertical direction and radius information in the horizontal direction.
If `dst` is specified it is used to store the polar image. If `dst` is not specified, its size is automatically determined. Variable `interpolation` specifies type of interpolation to be used. Valid values include *simple* (the default) or *bilinear* interpolation. Variable `mode` determines whether the *full* image is converted to the polar space (implying empty regions in the polar image), or whether only the *valid* central part of the polar transform is returned (the default).

[res] image.logpolar([dst,], src, [interpolation], [mode])

Converts image `src` to log-polar coordinates. In the log-polar image, angular information is in the vertical direction and log-radius information in the horizontal direction.
If `dst` is specified it is used to store the polar image. If `dst` is not specified, its size is automatically determined. Variable `interpolation` specifies type of interpolation to be used. Valid values include *simple* (the default) or *bilinear* interpolation. Variable `mode` determines whether the *full* image is converted to the log-polar space (implying empty regions in the log-polar image), or whether only the *valid* central part of the log-polar transform is returned (the default).

[res] image.hflip([dst,] src)

Flips image `src` horizontally (left<->right). If `dst` is provided, it is used to store the output image. Otherwise, returns a new `res` Tensor.

[res] image.vflip([dst,], src)

Flips image `src` vertically (upsize<->down). If `dst` is provided, it is used to store the output image. Otherwise, returns a new `res` Tensor.

[res] image.flip([dst,] src, flip_dim)

Flips image `src` along the specified dimension. If `dst` is provided, it is used to store the output image. Otherwise, returns a new `res` Tensor.

[res] image.minmax{tensor, [min, max, ...]}

Compresses image `tensor` between `min` and `max`.
When omitted, `min` and `max` are infered from `tensor:min()` and `tensor:max()`, respectively.
The `tensor` is normalized using `min` and `max` by performing :

```
tensor:add(-min):div(max-min)
```

Other optional arguments (`...`) include `symm`, `inplace`, `saturate`, and `tensorOut`.
When `symm=true` and `min` and `max` are both omitted, `max = min*2` in the above equation. This results in a symmetric dynamic range that is particularly useful for drawing filters. The default is `false`.
When `inplace=true`, the result of the compression is stored in `tensor`.
The default is `false`.
When `saturate=true`, the result of the compression is passed through a function that clips the values between 0 and 1 (i.e. anything below 0 is set to 0, anything above 1 is set to 1).
When provided, Tensor `tensorOut` is used to store results.
Note that arguments should be provided as key-value pairs (in a table).

[res] image.gaussianpyramid([dst,] src, scales)

Constructs a Gaussian pyramid of scales `scales` from a 2D or 3D `src` image or size `[nChannel x] width x height`. Each Tensor at index `i` in the returned list of Tensors has size `[nChannel x] width*scales[i] x height*scales[i]`.

If list `dst` is provided, with or without Tensors, it is used to store the output images. Otherwise, returns a new `res` list of Tensors.

Internally, this function makes use of functions image.gaussian, image.scale and image.convolve.

Simple Drawing Routines

This section includes simple routines to draw on images.

[res] image.drawText(src, text, x, y, [options])

Draws text onto a 3-channel Tensor (C x H x W) at the x-offset `x` and y-offset `y`.

The `options` table can be passed in to set color, background color, in-place etc.

Options:
* `color` - [table] The text color. A table of 3 numbers `{R, G, B}`, each number scaled between 0 and 255. For example, `red` is `{255, 0, 0}`
* bg - [table] The background color where text is drawn. Same format as color.
* size - [number] Size of the text to be drawn. `Default value = 1`.
* wrap - [boolean] If the text goes out of bounds, wrap it with a newline automatically. `default value = true`
* inplace - [boolean] If true, draws directly on the input tensor and returns it. `default value = false`

Example:

```
image.drawText(image.lena(), "hello\nworld", 10, 10)
image.drawText(image.lena(), "hello\nworld", 10, 20,{color = {0,
255, 0}, size = 5})
image.drawText(image.lena(), "hello\nworld", 10, 20,{color = {0,
255, 0}, bg = {255, 0, 0}, size = 5})
```

[res] image.drawRect(src, x1, y1, x2, y2, [options])

Draws a rectangle onto a 3-channel Tensor (C x H x W). The top-left corner of the rectangle is `x1, y1`, and the bottom-right corner is `x2, y2`.

The `options` table can be passed in to set color, in-place etc.

Options:
* `color` - [table] The rectangle color. A table of 3 numbers `{R, G, B}`, each

number scaled between 0 and 255. For example, `red` is `{255, 0, 0}`
* `lineWidth` - [number] The width of the rectangle line, in pixels
* `inplace` - [boolean] If true, draws directly on the input tensor and returns
it. `default value = false`

Example:

```
image.drawRect(image.lena(), 200, 200, 370, 400, {lineWidth = 5,
color = {0, 255, 0}})
```

Parameterized transformations

This section includes functions for performing transformations on
images requiring parameter Tensors like a warp `field` or a convolution
`kernel`.

[res] image.warp([dst,]src,field, [mode,offset,clamp_mode,pad_val])

Warps image `src` (of size `KxHxW`)
according to flow field `field`. The latter has size `2xHxW` where the
first dimension is for the `(y,x)` flow field. String `mode` can
take on values lanczos,
bicubic,
bilinear (the default),
or *simple*. When `offset` is true (the default), `(x,y)` is added to the flow field.
The `clamp_mode` variable specifies how to handle the interpolation of samples off the input image.
Permitted values are strings *clamp* (the default) or *pad*.
When `clamp_mode` equals `pad`, the user can specify the padding value with `pad_val`
(default = 0). Note: setting this value when `clamp_mode` equals `clamp` will result in an error.
If `dst` is specified, it is used to store the result of the warp.
Otherwise, returns a new `res` Tensor.

[res] image.affinetransform([dst,]src,matrix, [mode,translation,clamp_mode,pad_val])

Warps image `src` (of size `KxHxW`)
according to `(y,x)` affine transformation defined by `matrix`.
The latter has size `2x2`. String `mode` can
take on values lanczos,
bicubic,
bilinear (the default),
or *simple*.
Additional translation can be added to the image before affine transformation with
`translation`.(Default is `torch.Tensor{0, 0}`.)

The `clamp_mode` variable specifies how to handle the interpolation of samples off the input image.
Permitted values are strings *clamp* (the default) or *pad*.
When `clamp_mode` equals `pad`, the user can specify the padding value with `pad_val` (default = 0). Note: setting this value when `clamp_mode` equals `clamp` will result in an error.
If `dst` is specified, it is used to store the result of the warp.
Otherwise, returns a new `res` Tensor.

[res] image.convolve([dst,] src, kernel, [mode])

Convolves Tensor `kernel` over image `src`. Valid string values for argument `mode` are:
* *full*: the `src` image is effectively zero-padded such that the `res` of the convolution has the same size as `src`;
* *valid* (the default): the `res` image will have `math.ceil(kernel/2)` less columns and rows on each side;
* *same*: performs a *full* convolution, but crops out the portion fitting the output size of *valid*;
Note that this function internally uses
torch.conv2.
If `dst` is provided, it is used to store the output image.
Otherwise, returns a new `res` Tensor.

[res] image.lcn(src, [kernel])

Local contrast normalization (LCN) on a given `src` image using kernel `kernel`.
If `kernel` is not given, then a default `9x9` Gaussian is used
(see image.gaussian).

To prevent border effects, the image is first global contrast normalized
(GCN) by substracting the global mean and dividing by the global
standard deviation.

Then the image is locally contrast normalized using the following equation:

```
res = (src - lm(src)) / sqrt( lm(src) - lm(src*src) )
```

where `lm(x)` is the local mean of each pixel in the image (i.e.
`image.convolve(x,kernel)`) and `sqrt(x)` is the element-wise
square root of `x`. In other words, LCN performs

local substractive and divisive normalization.

Note that this implementation is different than the LCN Layer defined on page 3 of What is the Best Multi-Stage Architecture for Object Recognition?.

[res] image.erode(src, [kernel, pad])

Performs a morphological erosion
on binary (zeros and ones) image `src` using odd
dimensioned morphological binary kernel `kernel`.
The default is a kernel consisting of ones of size `3x3`. Number
`pad` is the value to assume outside the image boundary when performing
the convolution. The default is 1.

[res] image.dilate(src, [kernel, pad])

Performs a morphological dilation
on binary (zeros and ones) image `src` using odd
dimensioned morphological binary kernel `kernel`.
The default is a kernel consisting of ones of size `3x3`. Number
`pad` is the value to assume outside the image boundary when performing
the convolution. The default is 0.

Graphical User Interfaces

The following functions, except for image.toDisplayTensor,
require package qtlua and can only be
accessed via the `qlua` Lua interpreter (as opposed to the
th or luajit interpreter).

[res] image.toDisplayTensor(input, [...])

Optional arguments `[...]` expand to `padding`, `nrow`, `scaleeach`, `min`, `max`,
`symmetric`, `saturate`.
Returns a single `res` Tensor that contains a grid of all in the images in `input`.
The latter can either be a table of image Tensors of size `height x width` (greyscale) or
`nChannel x height x width` (color),
or a single Tensor of size `batchSize x nChannel x height x width` or `nChannel x height x width`
where `nChannel=[3,1]`, `batchSize x height x width` or `height x width`.

When `scaleeach=false` (the default), all detected images
are compressed with successive calls to image.minmax:

```
image.minmax{tensor=input[i], min=min, max=max, symm=symmetric,
saturate=saturate}
```

`padding` specifies the number of padding pixels between images. The default is 0.
`nrow` specifies the number of images per row. The default is 6.

Note that arguments can also be specified as key-value arguments (in a table).

[res] image.display(input, [...])

Optional arguments `[...]` expand to `zoom`, `min`, `max`, `legend`, `win`,
`x`, `y`, `scaleeach`, `gui`, `offscreen`, `padding`, `symm`, `nrow`.
Displays `input` image(s) with optional saturation and zooming.
The `input`, which is either a Tensor of size `HxW`, `KxHxW` or `Kx3xHxW`, or list,
is first prepared for display by passing it through image.toDisplayTensor:

```
input = image.toDisplayTensor{
   input=input, padding=padding, nrow=nrow, saturate=saturate,
   scaleeach=scaleeach, min=min, max=max, symmetric=symm
}
```

The resulting `input` will be displayed using qtlua.
The displayed image will be zoomed by a factor of `zoom`. The default is 1.
If `gui=true` (the default), the graphical user inteface (GUI)
is an interactive window that provides the user with the ability to zoom in or out.
This can be turned off for a faster display. `legend` is a legend to be displayed,
which has a default value of `image.display`. `win` is an optional qt window descriptor.
If `x` and `y` are given, they are used to offset the image. Both default to 0.
When `offscreen=true`, rendering (to generate images) is performed offscreen.

[window, painter] image.window([...])

Creates a window context for images.
Optional arguments `[...]` expand to `hook_resize`, `hook_mousepress`,
`hook_mousedoublepress`.
These have a default value of `nil`, but may correspond to commensurate qt objects.

Color Space Conversions

This section includes functions for performing conversions between different color spaces.

[res] image.rgb2lab([dst,] src)

Converts a `src` RGB image to Lab.
If `dst` is provided, it is used to store the output image. Otherwise, returns a new `res` Tensor.

[res] image.lab2rgb([dst,] src)

Converts a `src` Lab image to RGB.
If `dst` is provided, it is used to store the output image. Otherwise, returns a new `res` Tensor.

[res] image.rgb2yuv([dst,] src)

Converts a RGB image to YUV. If `dst` is provided, it is used to store the output image. Otherwise, returns a new `res` Tensor.

[res] image.yuv2rgb([dst,] src)

Converts a YUV image to RGB. If `dst` is provided, it is used to store the output image. Otherwise, returns a new `res` Tensor.

[res] image.rgb2y([dst,] src)

Converts a RGB image to Y (discard U and V).
If `dst` is provided, it is used to store the output

image. Otherwise, returns a new `res` Tensor.

[res] image.rgb2hsl([dst,] src)

Converts a RGB image to HSL.
If `dst` is provided, it is used to store the output
image. Otherwise, returns a new `res` Tensor.

[res] image.hsl2rgb([dst,] src)

Converts a HSL image to RGB.
If `dst` is provided, it is used to store the output
image. Otherwise, returns a new `res` Tensor.

[res] image.rgb2hsv([dst,] src)

Converts a RGB image to HSV.
If `dst` is provided, it is used to store the output
image. Otherwise, returns a new `res` Tensor.

[res] image.hsv2rgb([dst,] src)

Converts a HSV image to RGB.
If `dst` is provided, it is used to store the output
image. Otherwise, returns a new `res` Tensor.

[res] image.rgb2nrgb([dst,] src)

Converts an RGB image to normalized-RGB.

[res] image.y2jet([dst,] src)

Converts a L-levels (1 to L) greyscale image into a L-levels jet heat-map.
If `dst` is provided, it is used to store the output image. Otherwise, returns a new `res` Tensor.

This is particulary helpful for understanding the magnitude of the values of a matrix, or easily spot peaks in scalar field (like probability densities over a 2D area).
For example, you can run it as

```
image.display{image=image.y2jet(torch.linspace(1,10,10)), zoom=50}
```

Common Operations

It is possible to manage multiple plots at a time, printing plots to png, eps or pdf files or creating plots directly on png or eps or pdf files.

There are also several handy operations for decorating plots which are common to many of the plotting functions.

gnuplot.setgnuplotexe(exe)

Manually set the location of gnuplot executable.

gnuplot.setterm(teerm)

Manually set the gnuplot terminal.

gnuplot.closeall()

Close all gnuplot active connections. This will not be able to close open windows, since on the backend gnuplot also can not close windows.

gnuplot.figure([n])

Select (or create) a new figure with id `n`. Note that until a plot command is given, the window will not be created. When `n` is skipped, a new figure is created with the next consecutive id.

gnuplot.xlabel(label)

Sets the label of `x` axis to `label`. Only supported for gnuplot version 4.4 and above.

gnuplot.ylabel(label)

Sets the label of `y` axis to `label`. Only supported for gnuplot version 4.4 and above.

gnuplot.zlabel(label)

Sets the label of `z` axis to `label`. Only supported for gnuplot version 4.4 and above.

gnuplot.title(title)

Sets the title of the plot to `title`. Only supported for gnuplot version 4.4 and above.

gnuplot.grid(toggle)

If `toggle` is `true` then a grid is displayed, else it is hidden. Only supported for gnuplot version 4.4 and above.

gnuplot.movelegend(hloc,vloc)

Set the location of legend key. `hloc` can be `left`, `right` or `center`. `vloc` can be `top`, `bottom` or `middle`. Only supported for gnuplot version 4.4 and above.

gnuplot.axis(axis)

Sets the properties of axis for the current plot.

- `auto` : auto scales the axis to fit data and plot canvas
- `image` : scales the axis aspect ratio so that a circle is drawn as circle.

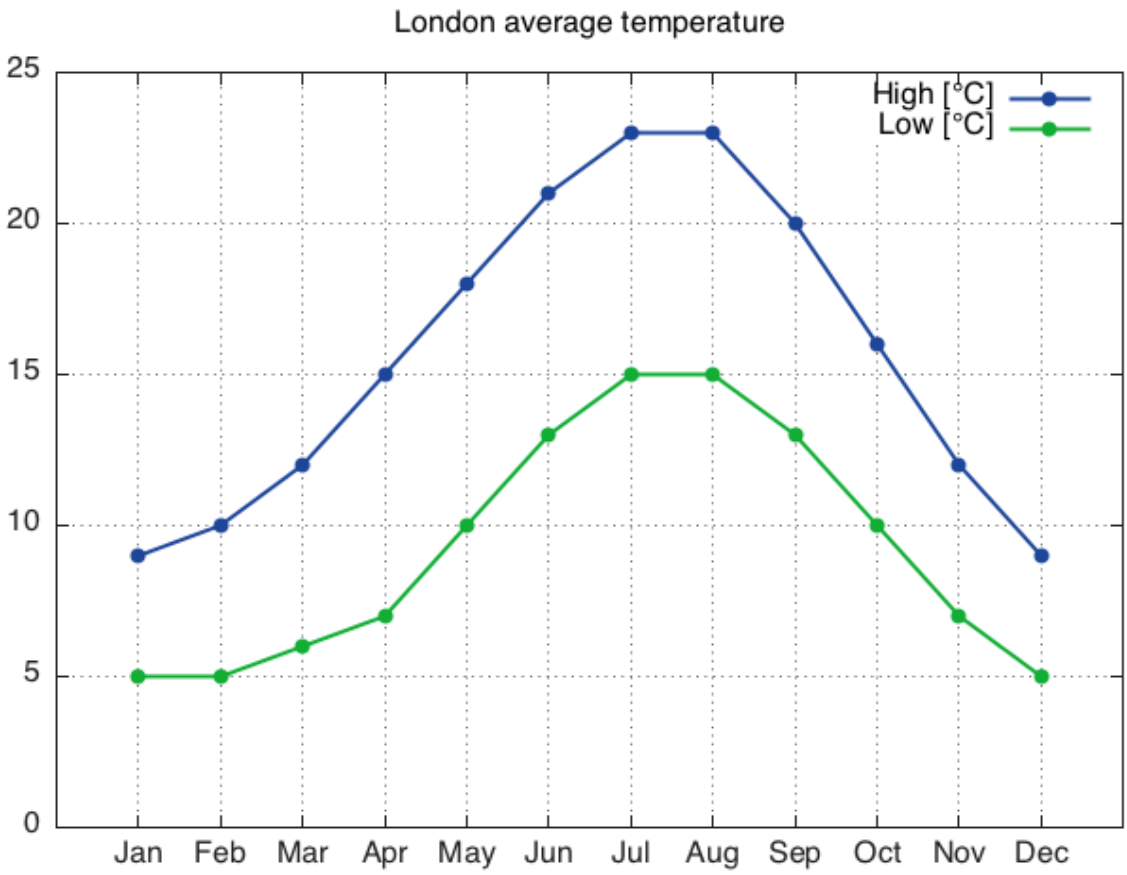
- `equal` : same as `image`.
- `fill` : resets the aspect ratio of the plot to original values so that it fills up the canvas as good as possible.
- `{xmin,xmax,ymin,ymax}` : Sets the limits of x and y axes. Use an empty string (2 apostophes in a row) if you want to keep the current value.

gnuplot.raw(command)

This command is useful for advanced users of gnuplot. `command` is directly passed to gnuplot without any formatting.

Let's see an example, by plotting labels for the `xtic`

```
LondonTemp = torch.Tensor{{9, 10, 12, 15, 18, 21, 23, 23, 20, 16,
12, 9},
                          {5,  5,  6,  7, 10, 13, 15, 15, 13, 10,
7, 5}}
gnuplot.plot({'High [°C]',LondonTemp[1]},{'Low
[°C]',LondonTemp[2]})
gnuplot.raw('set xtics ("Jan" 1, "Feb" 2, "Mar" 3, "Apr" 4, "May"
5, "Jun" 6, "Jul" 7, "Aug" 8, "Sep" 9, "Oct" 10, "Nov" 11, "Dec"
12)')
gnuplot.plotflush()
gnuplot.axis{0,13,0,''}
gnuplot.grid(true)
gnuplot.title('London average temperature')
```



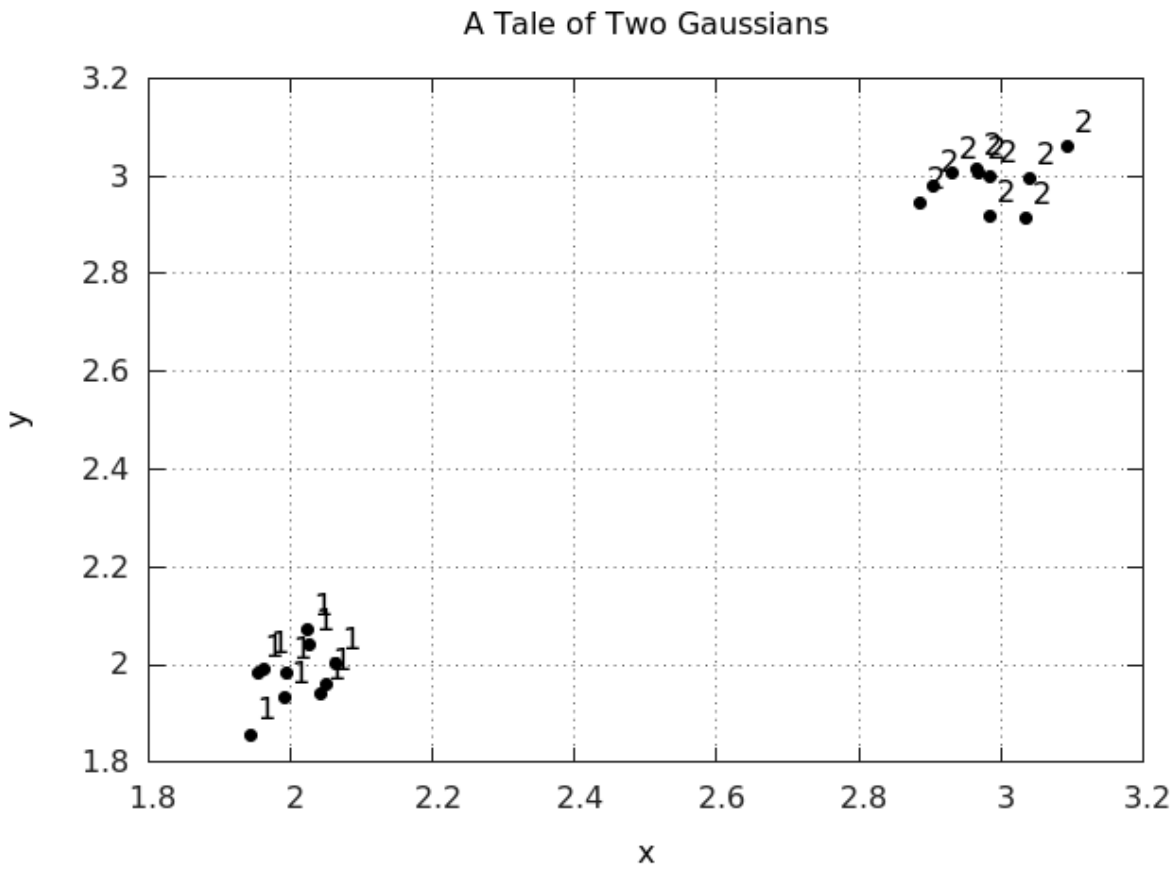
We can show another example of its usage by labeling data points. Here, we generate two clusters of data with two Gaussian distributions, write it to a text file in three columns of (label, x_coor, y_coor), and plot the points with labels.

```
labels_tab = {}
for i=1,20 do
  table.insert(labels_tab, math.ceil(i/10))
end
x = torch.cat(torch.Tensor(10):normal(2, 0.05),
torch.Tensor(10):normal(3, 0.05), 1)
y = torch.cat(torch.Tensor(10):normal(2, 0.05),
torch.Tensor(10):normal(3, 0.05), 1)

file = io.open('gaussians.txt', 'w')
io.output(file)
for i=1,20 do
  io.write(string.format('%d %f %f\n', labels_tab[i], x[i], y[i]))
end
io.close(file)

gnuplot.pngfigure('plot_labels.png')
```

```
gnuplot.title('A Tale of Two Gaussians')
gnuplot.raw("plot 'gaussians.txt' using 2:3:(sprintf('%d', $1))
with labels point pt 7 offset char 0.5,0.5 notitle")
gnuplot.xlabel('x')
gnuplot.ylabel('y')
gnuplot.grid(true)
gnuplot.plotflush()
```



Saving Plots to Files

Any of the above plotting utilities can also be used for directly plotting into `eps` or `png` files, or `pdf` files if your gnuplot installation allows. A final gnuplot.plotflush() command ensures that all output is written to the file properly.

```
gnuplot.epsfigure('test.eps')
gnuplot.plot({'Sin Curve',torch.sin(torch.linspace(-5,5))})
gnuplot.xlabel('X')
gnuplot.ylabel('Y')
gnuplot.plotflush()
```

gnuplot.epsfigure(fname)

Creates a figure directly on the `eps` file given with `fname`. This uses `Gnuplot` terminal `postscript eps enhanced color`.

gnuplot.pdffigure(fname)

Only available if your installation of gnuplot has been compiled with `pdf` support enabled.

Creates a figure directly on the `pdf` file given with `fname`. This uses `Gnuplot` terminal `pdf enhanced color`, or `pdfcairo enhanced color` if available.

gnuplot.pngfigure(fname)

Creates a figure directly on the `png` file given with `fname`. This uses `Gnuplot` terminal `png`, or `pngcairo` if available.

gnuplot.svgfigure(fname)

Creates a figure directly on the `svg` file given with `fname`. This uses `Gnuplot` terminal `svg`.

gnuplot.figprint(fname)

Prints the current figure to the given file with name `fname`. Only `png` or `eps` files are supported by default. If your gnuplot installation allows, `pdf` files are also supported.

gnuplot.plotflush([n])

This command sends `unset output` to underlying gnuplot. Useful for flushing file based terminals.

gnuplot.close()

Closes open file handles. Prevents too many handles staying open if creating lots of plots.

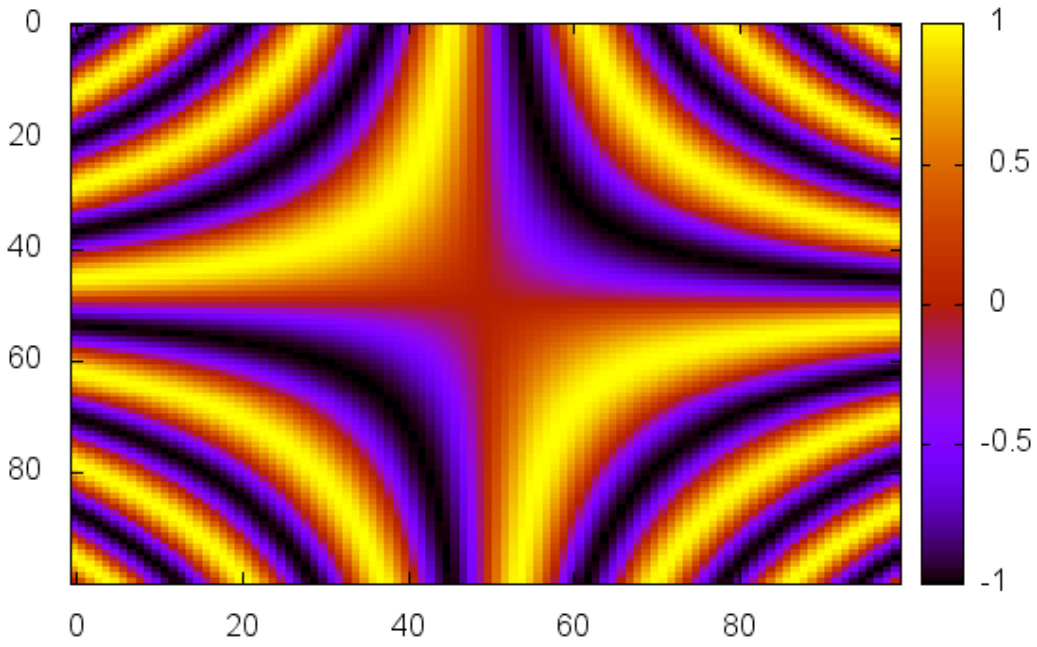
Plotting Matrices

A given matrix can be plotted using 2D contour plot on a surface.

gnuplot.imagesc(z, [‘color’ or ‘gray’])

Plot surface `z` using contour plot. The second argument defines the color palette for the display. By default, grayscale colors are used, however, one can also use any color palette available in `Gnuplot`.

```
x = torch.linspace(-1,1)
xx = torch.Tensor(x:size(1),x:size(1)):zero():addr(1,x,x)
xx = xx*math.pi*6
gnuplot.imagesc(torch.sin(xx),'color')
```



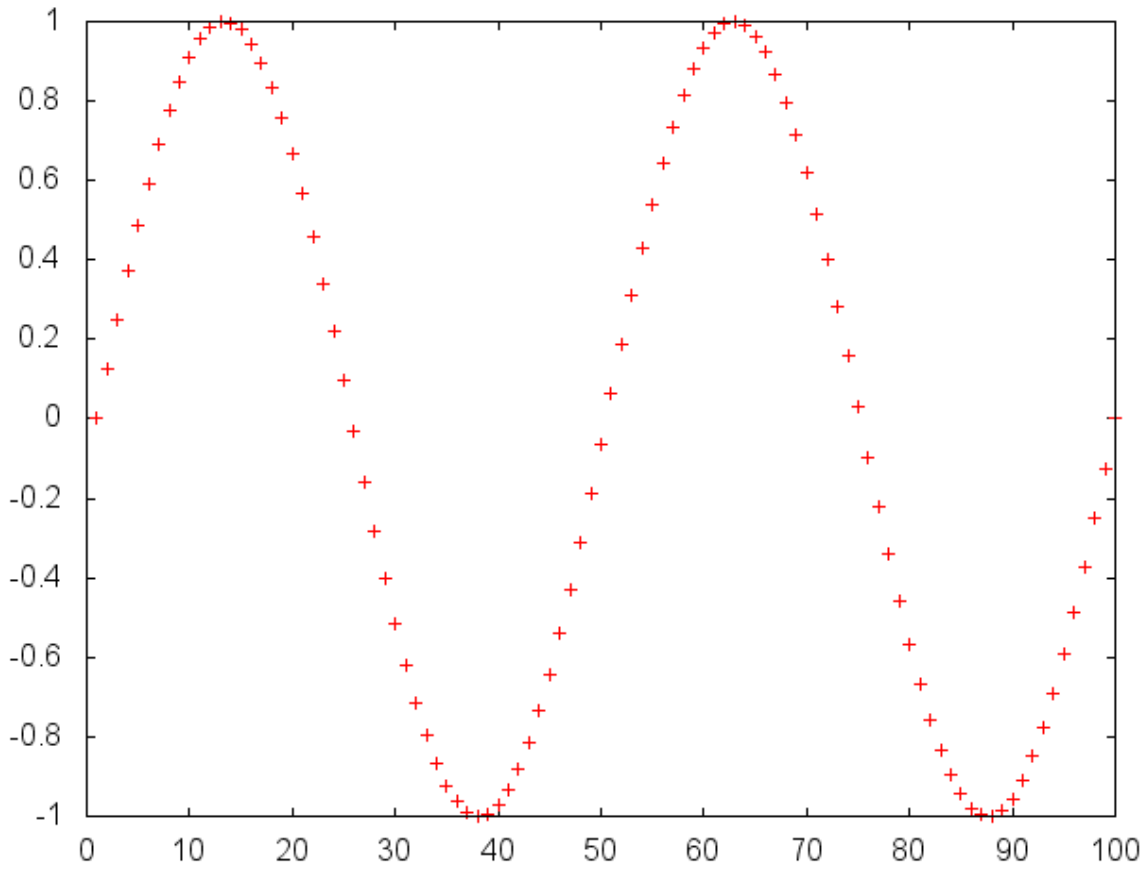
Plotting Lines

Line plotting functionality covers many configurations from simplest case of plotting a single vector to displaying multiple lines at once with custom line specifictions.

gnuplot.plot(x)

Plot vector `x` using dots of first default `Gnuplot` type.

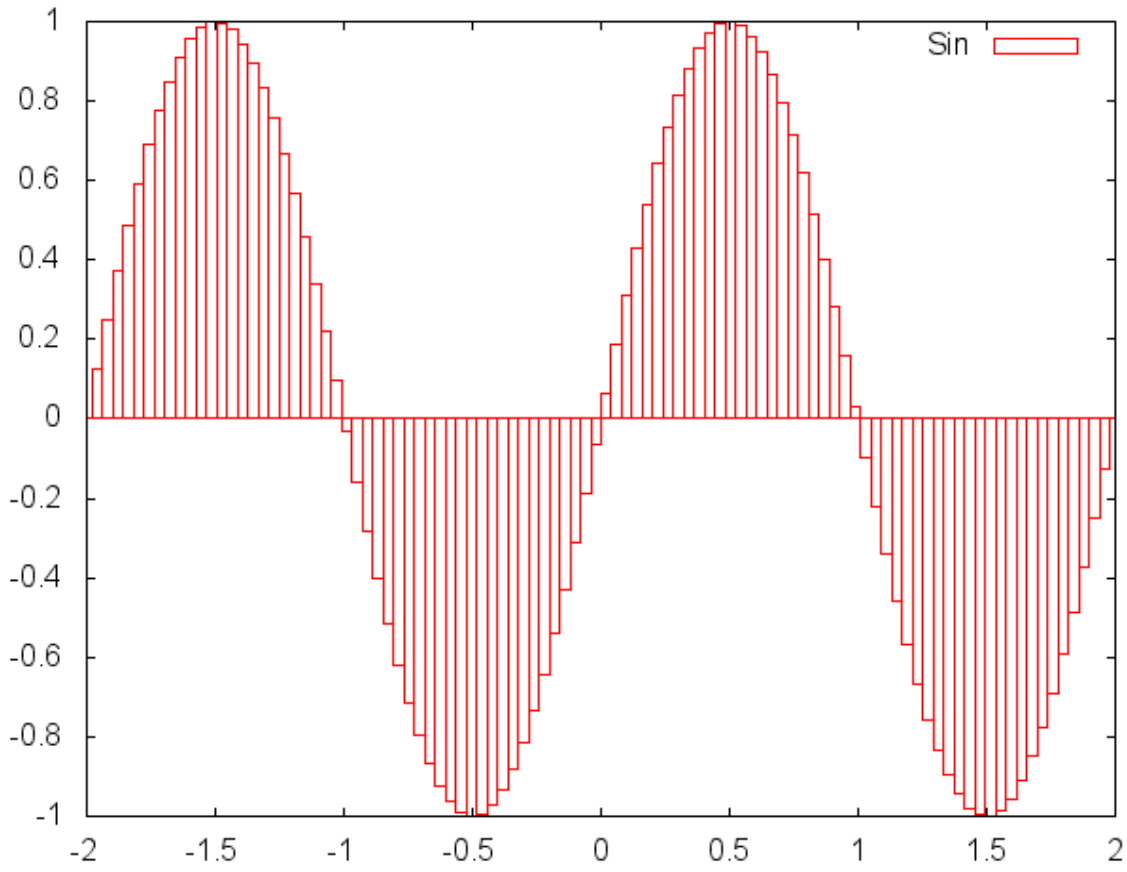
```
x=torch.linspace(-2*math.pi,2*math.pi)
gnuplot.plot(torch.sin(x))
```



In more general form, plot vector `y` vs `x` using the format specified. The possible entries of format string can be:
* `.` for dots

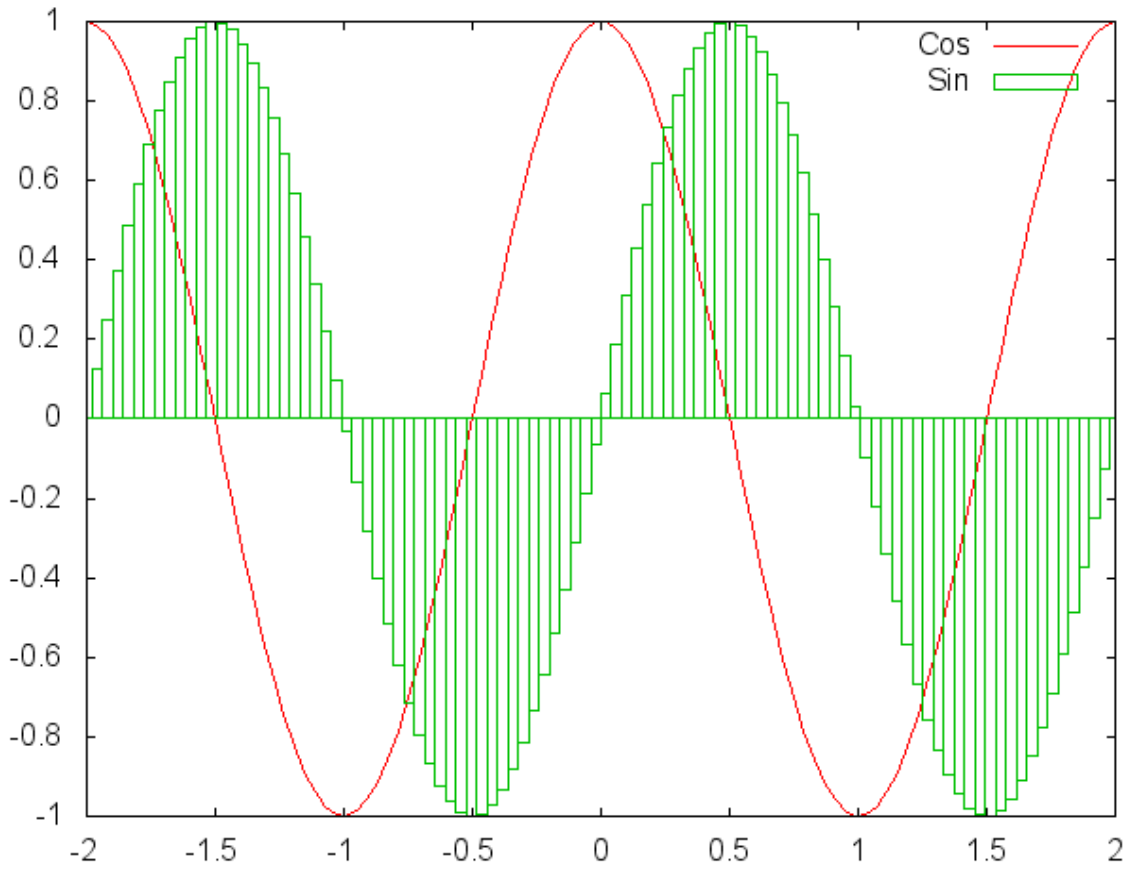
* `+` for points
* `-` for lines
* `+-` for points and lines
* `~` for using smoothed lines with cubic interpolation
* `|` for using boxes
* `v` for drawing vector fields. (In this case, `x` and `y` have to be two column vectors `(x, xdelta)`, `(y, ydelta)`)
* custom string, one can also pass custom strings to use full capability of gnuplot.

```
x = torch.linspace(-2*math.pi,2*math.pi)
gnuplot.plot('Sin',x/math.pi,torch.sin(x),'|')
```



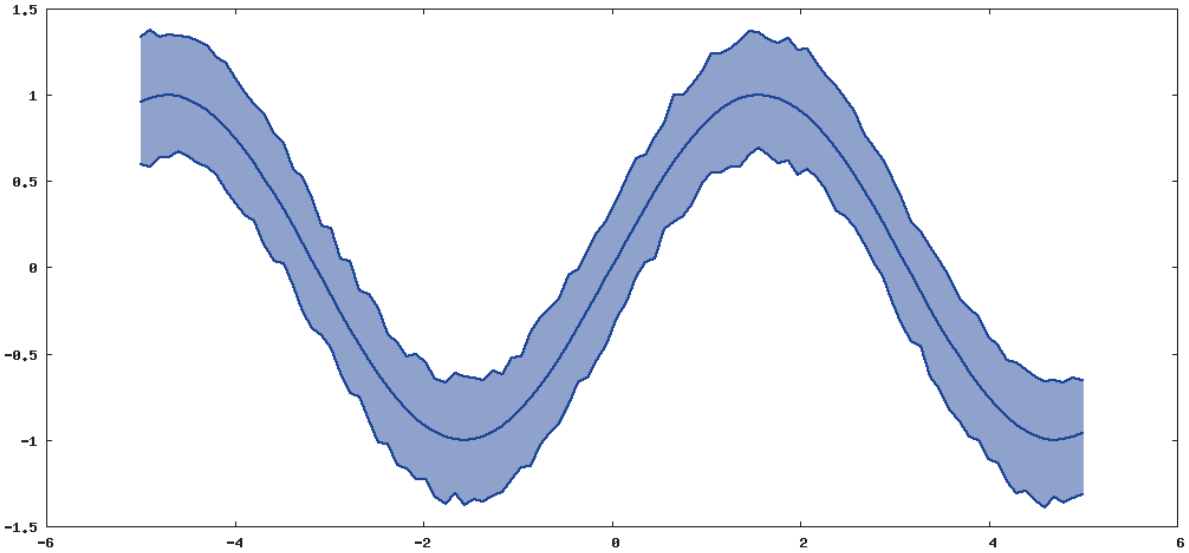
To plot multiple curves at a time, one can pass each plot struct in a table.

```
x = torch.linspace(-2*math.pi,2*math.pi)
gnuplot.plot({'Cos',x/math.pi,torch.cos(x),'~'},
{'Sin',x/math.pi,torch.sin(x),'|'})
```



One can pass data with multiple columns and use custom gnuplot style strings too. When multi-column data
is used, the first column is assumed to be the `x` values and the rest of the columns are separate `y` series.

```
x = torch.linspace(-5,5)
y = torch.sin(x)
yp = y+0.3+torch.rand(x:size())*0.1
ym = y-(torch.rand(x:size())*0.1+0.3)
yy = torch.cat(x,ym,2)
yy = torch.cat(yy,yp,2)
gnuplot.plot({yy,'with filledcurves fill transparent solid 0.5'},
{x,yp,'with lines ls 1'},{x,ym,'with lines ls 1'},{x,y,'with lines
ls 1'})
```


1.5
1
0.5
0
-0.5
-1
-1.5
-6
-4
-2
0
2
4
6

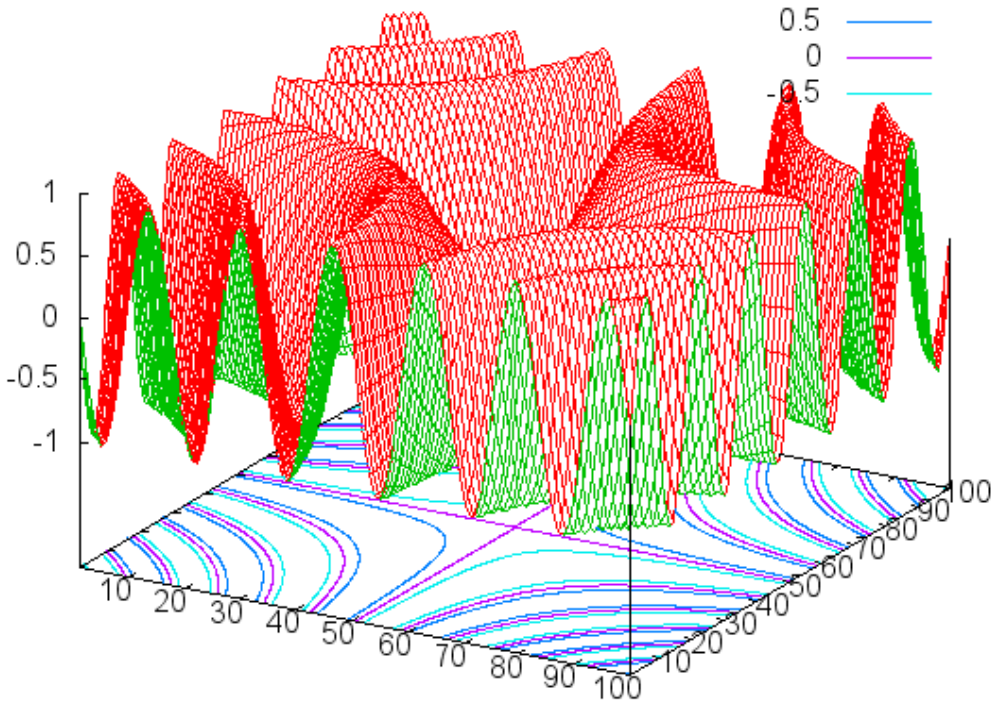
Plotting 3D Surfaces

Surface plotting creates a 3D surface plot of a given matrix `z`. Entries of `z` are used as height values. It is also possible to specify `x` and `y` locations corresponding to each point in `z`. If a terminal with interactive capabilities is being used by `Gnuplot` backend (like `x11` or `wxt` or `qt`), then rotating, zooming is also possible.

gnuplot.splot(z)

Plot surface `z` in 3D.

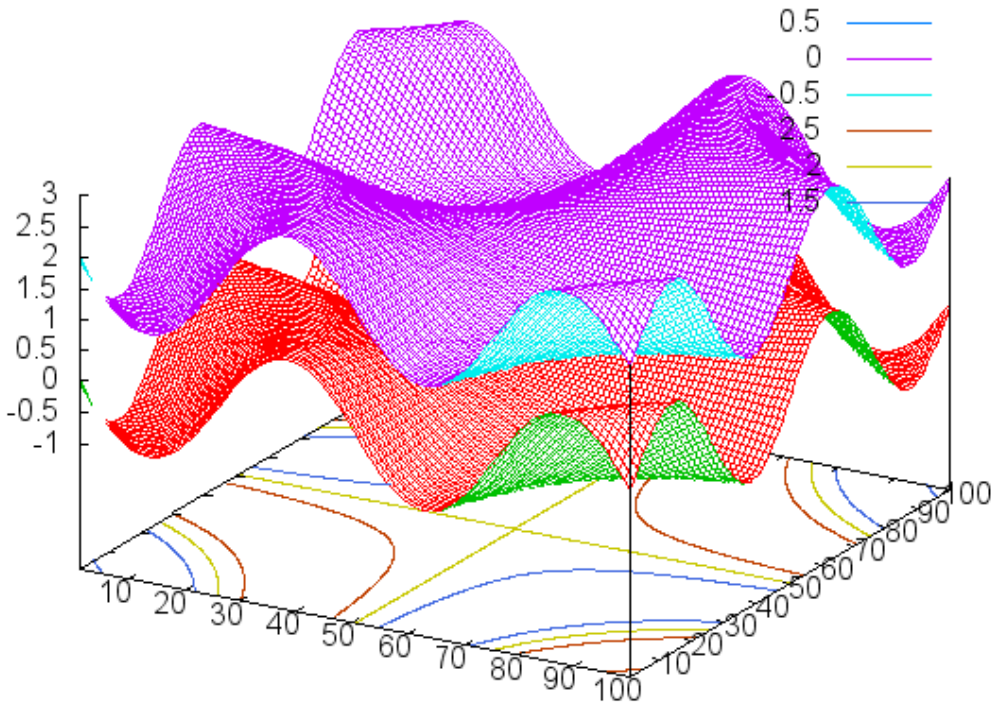
```
x = torch.linspace(-1,1)
xx = torch.Tensor(x:size(1),x:size(1)):zero():addr(1,x,x)
xx = xx*math.pi*6
gnuplot.splot(torch.sin(xx))
```



It is also possible to specify the `x` and `y` locations of each point in `z` by `gnuplot.splot(x,y,z)`. In this `x` and `y` has to be the same shape as `z`.

One can also display multiple surfaces at a time.

```
x = torch.linspace(-1,1)
xx = torch.Tensor(x:size(1),x:size(1)):zero():addr(1,x,x)
xx = xx*math.pi*2
gnuplot.splot({torch.sin(xx)},{torch.sin(xx)+2})
```



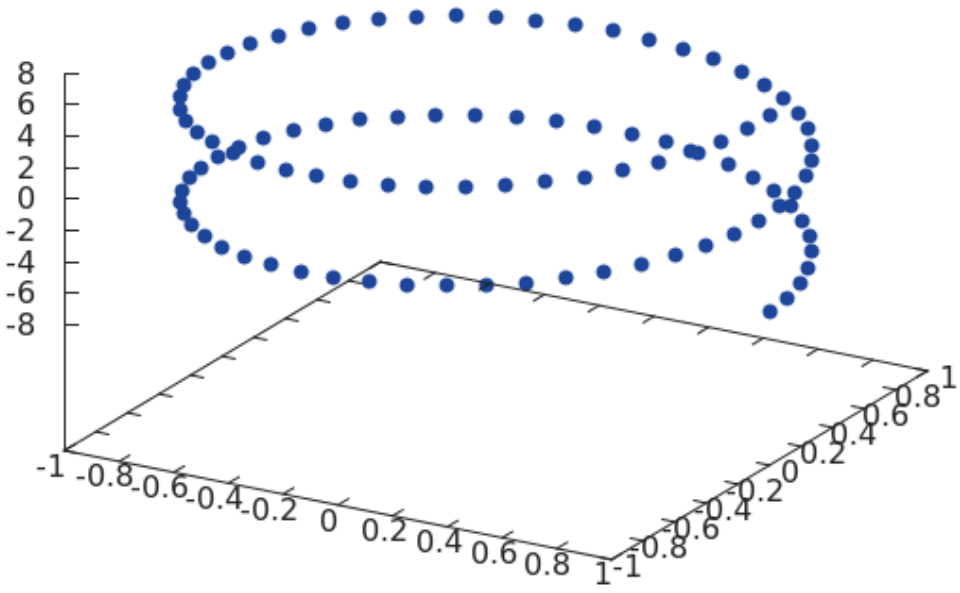
Plotting 3D Points

Arbitrary 3D point constellations can be plotted using an API similar to the `scatter3` function in Matalb.

gnuplot.scatter3(x, y, z)

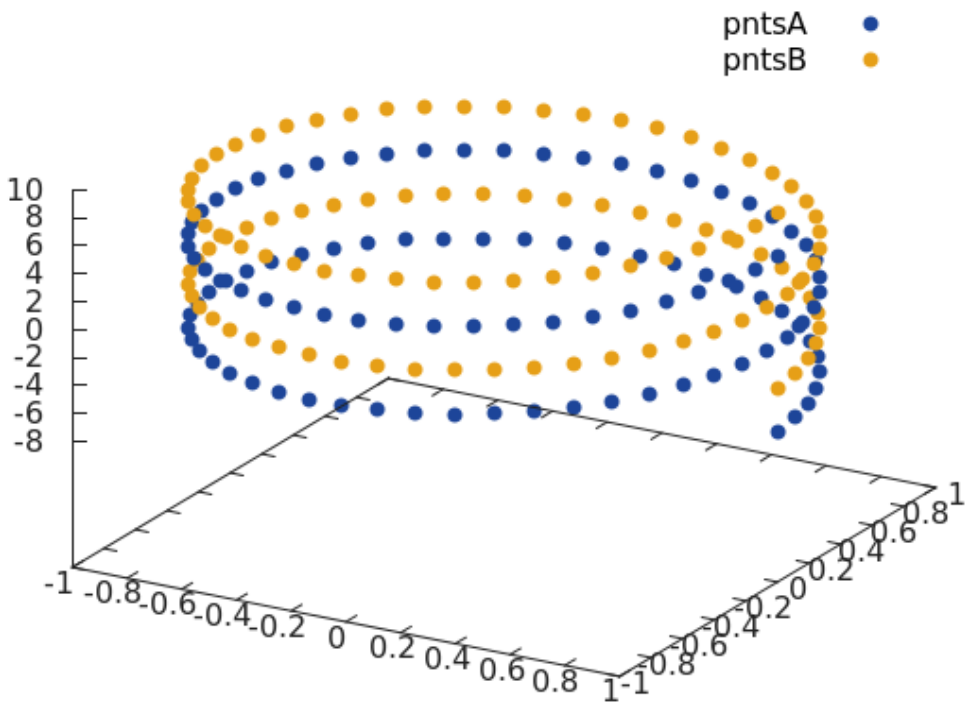
Plot `(x_i, y_i, z_i)` triplets in 3D.

```
z = torch.linspace(-2 * math.pi, 2 * math.pi)
x = z:clone():cos()
y = z:clone():sin()
gnuplot.scatter3(x, y, z)
```



It is also possible to specify a header, as well as multiple scatter plot sets on the same axis.

```
z1 = torch.linspace(-2 * math.pi, 2 * math.pi)
x = z1:clone():cos()
y = z1:clone():sin()
z2 = z1:clone():add(math.pi)
gnuplot.scatter3({'pntsA', x, y, z1}, {'pntsB', x, y, z2})
```



Histograms

Given a tensor, the distribution of values can be plotted using `gnuplot.hist` function.

gnuplot.hist(x, [nbins, min, max])

Plot the histogram of values in `N-D` tensor `x`, optionally using `nbins` number of bins and only using values between `min` and `max`.

```
gnuplot.hist(torch.randn(100000),100)
```