PyTorch Documentation

0.1.11_5

Torch Contributors

4 06, 2017

Notes

PyTorch is an optimized tensor library for deep learning using GPUs and CPUs.

CHAPTER 1

Autograd mechanics

This note will present an overview of how autograd works and records the operations. It's not strictly necessary to understand all this, but we recommend getting familiar with it, as it will help you write more efficient, cleaner programs, and can aid you in debugging.

Excluding subgraphs from backward

Every Variable has two flags: `requires_grad` and `volatile`. They both allow for fine grained exclusion of subgraphs from gradient computation and can increase efficiency.

`requires_grad`

If there's a single input to an operation that requires gradient, its output will also require gradient. Conversely, only if all inputs don't require gradient, the output also won't require it. Backward computation is never performed in the subgraphs, where all Variables didn't require gradients.

```
>>> x = Variable(torch.randn(5, 5))
>>> y = Variable(torch.randn(5, 5))
>>> z = Variable(torch.randn(5, 5), requires_grad=True)
>>> a = x + y
>>> a.requires_grad
False
>>> b = a + z
>>> b.requires_grad
True
```

This is especially useful when you want to freeze part of your model, or you know in advance that you're not going to use gradients w.r.t. some parameters. For example if you want to finetune a pretrained CNN, it's enough to switch the `requires_grad` flags in the frozen base, and no intermediate buffers will be saved, until the computation gets to the last layer, where the affine transform will use weights that require gradient, and the output of the network will also require them.

```
model = torchvision.models.resnet18(pretrained=True)
for param in model.parameters():
    param.requires_grad = False
# Replace the last fully-connected layer
# Parameters of newly constructed modules have requires_grad=True by default
model.fc = nn.Linear(512, 100)

# Optimize only the classifier
optimizer = optim.SGD(model.fc.parameters(), lr=1e-2, momentum=0.9)
```

volatile

Volatile is recommended for purely inference mode, when you're sure you won't be even calling *.backward()*. It's more efficient than any other autograd setting - it will use the absolute minimal amount of memory to evaluate the model. `volatile` also determines that `requires_grad is False`.

Volatile differs from *requires_grad* in how the flag propagates. If there's even a single volatile input to an operation, its output is also going to be volatile. Volatility spreads accross the graph much easier than non-requiring gradient - you only need a **single** volatile leaf to have a volatile output, while you need **all** leaves to not require gradient to have an output the doesn't require gradient. Using volatile flag you don't need to change any settings of your model parameters to use it for inference. It's enough to create a volatile input, and this will ensure that no intermediate states are saved.

```
>>> regular_input = Variable(torch.randn(5, 5))
>>> volatile_input = Variable(torch.randn(5, 5), volatile=True)
>>> model = torchvision.models.resnet18(pretrained=True)
>>> model(regular_input).requires_grad
True
>>> model(volatile_input).requires_grad
False
>>> model(volatile_input).volatile
True
>>> model(volatile_input).creator is None
True
```

How autograd encodes the history

Each Variable has a `.creator` attribute, that points to the function, of which it is an output. This is an entry point to a directed acyclic graph (DAG) consisting of `Function` objects as nodes, and references between them being the edges. Every time an operation is performed, a new `Function` representing it is instantiated, its *`forward()`* method is called, and its output `Variable` s creators are set to it. Then, by following the path from any `Variable` to the leaves, it is possible to reconstruct the sequence of operations that has created the data, and automatically compute the gradients.

An important thing to note is that the graph is recreated from scratch at every iteration, and this is exactly what allows for using arbitrary Python control flow statements, that can change the overall shape and size of the graph at every iteration. You don't have to encode all possible paths before you launch the training - what you run is what you differentiate.

In-place operations on Variables

Supporting in-place operations in autograd is a hard matter, and we discourage their use in most cases. Autograd's aggressive buffer freeing and reuse makes it very efficient and there are very few occasions when in-place operations actually lower memory usage by any significant amount. Unless you're operating under heavy memory pressure, you might never need to use them.

There are two main reasons that limit the applicability of in-place operations:

1. Overwriting values required to compute gradients. This is why variables don't support `log_`. Its gradient formula requires the original input, and while it is possible to recreate it by computing the inverse operation, it is numerically unstable, and requires additional work that often defeats the purpose of using these functions.
2. Every in-place operation actually requires the implementation to rewrite the computational graph. Out-of-place versions simply allocate new objects and keep references to the old graph, while in-place operations, require changing the creator of all inputs to the `Function` representing this operation. This can be tricky, especially if there are many Variables that reference the same storage (e.g. created by indexing or transposing), and in-place functions will actually raise an error if the storage of modified inputs is referenced by any other `Variable`.

In-place correctness checks

Every variable keeps a version counter, that is incremented every time it's marked dirty in any operation. When a Function saves any tensors for backward, a version counter of their containing Variable is saved as well. Once you access `self.saved_tensors` it is checked, and if it's greater than the saved value an error is raised.

CHAPTER 2

CUDA semantics

torch.cuda keeps track of currently selected GPU, and all CUDA tensors you allocate will be created on it. The selected device can be changed with a *torch.cuda.device* context manager.

However, once a tensor is allocated, you can do operations on it irrespectively of your selected device, and the results will be always placed in on the same device as the tensor.

Cross-GPU operations are not allowed by default, with the only exception of *copy_()*. Unless you enable peer-to-peer memory accesses any attempts to launch ops on tensors spread across different devices will raise an error.

Below you can find a small example showcasing this:

```
x = torch.cuda.FloatTensor(1)
# x.get_device() == 0
y = torch.FloatTensor(1).cuda()
# y.get_device() == 0

with torch.cuda.device(1):
    # allocates a tensor on GPU 1
    a = torch.cuda.FloatTensor(1)

    # transfers a tensor from CPU to GPU 1
    b = torch.FloatTensor(1).cuda()
    # a.get_device() == b.get_device() == 1

    c = a + b
    # c.get_device() == 1

    z = x + y
    # z.get_device() == 0

    # even within a context, you can give a GPU id to the .cuda call
    d = torch.randn(2).cuda(2)
    # d.get_device() == 2
```

Best practices

Use pinned memory buffers

Host to GPU copies are much faster when they originate from pinned (page-locked) memory. CPU tensors and storages expose a *pin_memory()* method, that returns a copy of the object, with data put in a pinned region.

Also, once you pin a tensor or storage, you can use asynchronous GPU copies. Just pass an additional `async=True` argument to a *cuda()* call. This can be used to overlap data transfers with computation.

You can make the *DataLoader* return batches placed in pinned memory by passing `pin_memory=True` to its constructor.

Use nn.DataParallel instead of multiprocessing

Most use cases involving batched input and multiple GPUs should default to using *DataParallel* to utilize more than one GPU. Even with the GIL, a single python process can saturate multiple GPUs.

As of version 0.1.9, large numbers of GPUs (8+) might not be fully utilized. However, this is a known issue that is under active development. As always, test your use case.

There are significant caveats to using CUDA models with *multiprocessing*; unless care is taken to meet the data handling requirements exactly, it is likely that your program will have incorrect or undefined behavior.

CHAPTER 3

Extending PyTorch

In this note we'll cover ways of extending *torch.nn*, *torch.autograd*, and writing custom C extensions utilizing our C libraries.

Extending `torch.autograd`

Adding operations to *autograd* requires implementing a new *Function* subclass for each operation. Recall that *Function* s are what *autograd* uses to compute the results and gradients, and encode the operation history. Every new function requires you to implement 3 methods:

- `__init__` (*optional*) - if your operation is parametrized by/uses objects different than *Variable* s, you should pass them as arguments to `__init__`. For example, `AddConstant` function takes a scalar to add, while `Transpose` requires specifying which two dimensions to swap. If your function doesn't require any additional parameters, you can skip it.
- *forward()* - the code that performs the operation. It can take as many arguments as you want, with some of them being optional, if you specify the default values. Keep in mind that only *Variable* s will be passed in here. You can return either a single *Variable* output, or a `tuple` of *Variable* s if there are multiple. Also, please refer to the docs of *Function* to find descriptions of useful methods that can be called only from *forward()*.
- *backward()* - gradient formula. It will be given as many arguments as there were outputs, with each of them representing gradient w.r.t. that output. It should return as many `Tensor` s as there were inputs, with each of them containing the gradient w.r.t. corresponding input. If you inputs didn't require gradient (see `needs_input_grad`), or it was non-differentiable, you can return `None`. Also, if you have optional arguments to `forward()` you can return more gradients than there were inputs, as long as they're all `None`.

Below you can find code for a `Linear` function from *torch.nn*, with additional comments:

```
# Inherit from Function
class Linear(Function):

    # bias is an optional argument
    def forward(self, input, weight, bias=None):
```

```
        self.save_for_backward(input, weight, bias)
        output = input.mm(weight.t())
        if bias is not None:
            output += bias.unsqueeze(0).expand_as(output)
        return output

    # This function has only a single output, so it gets only one gradient
    def backward(self, grad_output):
        # This is a pattern that is very convenient - at the top of backward
        # unpack saved_tensors and initialize all gradients w.r.t. inputs to
        # None. Thanks to the fact that additional trailing Nones are
        # ignored, the return statement is simple even when the function has
        # optional inputs.
        input, weight, bias = self.saved_tensors
        grad_input = grad_weight = grad_bias = None

        # These needs_input_grad checks are optional and there only to
        # improve efficiency. If you want to make your code simpler, you can
        # skip them. Returning gradients for inputs that don't require it is
        # not an error.
        if self.needs_input_grad[0]:
            grad_input = grad_output.mm(weight)
        if self.needs_input_grad[1]:
            grad_weight = grad_output.t().mm(input)
        if bias is not None and self.needs_input_grad[2]:
            grad_bias = grad_output.sum(0).squeeze(0)

        return grad_input, grad_weight, grad_bias
```

Now, to make it easier to use these custom ops, we recommend wrapping them in small helper functions:

```
def linear(input, weight, bias=None):
    # First braces create a Function object. Any arguments given here
    # will be passed to __init__. Second braces will invoke the __call__
    # operator, that will then use forward() to compute the result and
    # return it.
    return Linear()(input, weight, bias)
```

You probably want to check if the backward method you implemented actually computes the derivatives of your function. It is possible by comparing with numerical approximations using small finite differences:

```
from torch.autograd import gradcheck

# gradchek takes a tuple of tensor as input, check if your gradient
# evaluated with these tensors are close enough to numerical
# approximations and returns True if they all verify this condition.
input = (Variable(torch.randn(20,20).double(), requires_grad=True),)
test = gradcheck.gradcheck(Linear(), input, eps=1e-6, atol=1e-4)
print(test)
```

Extending `torch.nn`

nn exports two kinds of interfaces - modules and their functional versions. You can extend it in both ways, but we recommend using modules for all kinds of layers, that hold any parameters or buffers, and recommend using a functional form parameter-less operations like activation functions, pooling, etc.

Adding a functional version of an operation is already fully covered in the section above.

Adding a `Module`

Since *nn* heavily utilizes *autograd*, adding a new *Module* requires implementing a *Function* that performs the operation and can compute the gradient. From now on let's assume that we want to implement a `Linear` module and we have the function implementated as in the listing above. There's very little code required to add this. Now, there are two functions that need to be implemented:

- `__init__` (*optional*) - takes in arguments such as kernel sizes, numbers of features, etc. and initializes parameters and buffers.
- *forward()* - instantiates a *Function* and uses it to perform the operation. It's very similar to a functional wrapper shown above.

This is how a `Linear` module can be implemented:

```
class Linear(nn.Module):
    def __init__(self, input_features, output_features, bias=True):
        self.input_features = input_features
        self.output_features = output_features

        # nn.Parameter is a special kind of Variable, that will get
        # automatically registered as Module's parameter once it's assigned
        # as an attribute. Parameters and buffers need to be registered, or
        # they won't appear in .parameters() (doesn't apply to buffers), and
        # won't be converted when e.g. .cuda() is called. You can use
        # .register_buffer() to register buffers.
        # nn.Parameters can never be volatile and, different than Variables,
        # they require gradients by default.
        self.weight = nn.Parameter(torch.Tensor(input_features, output_features))
        if bias:
            self.bias = nn.Parameter(torch.Tensor(output_features))
        else:
            # You should always register all possible parameters, but the
            # optional ones can be None if you want.
            self.register_parameter('bias', None)

        # Not a very smart way to initialize weights
        self.weight.data.uniform_(-0.1, 0.1)
        if bias is not None:
            self.bias.data.uniform_(-0.1, 0.1)

    def forward(self, input):
        # See the autograd section for explanation of what happens here.
        return Linear()(input, self.weight, self.bias)
```

Writing custom C extensions

Coming soon. For now you can find an example at GitHub.

CHAPTER 4

Multiprocessing best practices

torch.multiprocessing is a drop in replacement for Python's `multiprocessing` module. It supports the exact same operations, but extends it, so that all tensors sent through a `multiprocessing.Queue`, will have their data moved into shared memory and will only send a handle to another process.

: When a *Variable* is sent to another process, both the `Variable.data` and `Variable.grad.data` are going to be shared.

This allows to implement various training methods, like Hogwild, A3C, or any others that require asynchronous operation.

Sharing CUDA tensors

Sharing CUDA tensors between processes is supported only in Python 3, using a `spawn` or `forkserver` start methods. `multiprocessing` in Python 2 can only create subprocesses using `fork`, and it's not supported by the CUDA runtime.

: CUDA API requires that the allocation exported to other processes remains valid as long as it's used by them. You should be careful and ensure that CUDA tensors you shared don't go out of scope as long as it's necessary. This shouldn't be a problem for sharing model parameters, but passing other kinds of data should be done with care. Note that this restriction doesn't apply to shared CPU memory.

See also: *Use nn.DataParallel instead of multiprocessing*

Best practices and tips

Avoiding and fighting deadlocks

There are a lot of things that can go wrong when a new process is spawned, with the most common cause of deadlocks being background threads. If there's any thread that holds a lock or imports a module, and `fork` is called, it's very likely that the subprocess will be in a corrupted state and will deadlock or fail in a different way. Note that even if you don't, Python built in libraries do - no need to look further than `multiprocessing`. `multiprocessing.Queue` is actually a very complex class, that spawns multiple threads used to serialize, send and receive objects, and they can cause aforementioned problems too. If you find yourself in such situation try using a `multiprocessing.queues.SimpleQueue`, that doesn't use any additional threads.

We're trying our best to make it easy for you and ensure these deadlocks don't happen but some things are out of our control. If you have any issues you can't cope with for a while, try reaching out on forums, and we'll see if it's an issue we can fix.

Reuse buffers passed through a Queue

Remember that each time you put a *Tensor* into a `multiprocessing.Queue`, it has to be moved into shared memory. If it's already shared, it is a no-op, otherwise it will incur an additional memory copy that can slow down the whole process. Even if you have a pool of processes sending data to a single one, make it send the buffers back - this is nearly free and will let you avoid a copy when sending next batch.

Asynchronous multiprocess training (e.g. Hogwild)

Using *torch.multiprocessing*, it is possible to train a model asynchronously, with parameters either shared all the time, or being periodically synchronized. In the first case, we recommend sending over the whole model object, while in the latter, we advise to only send the *state_dict()*.

We recommend using `multiprocessing.Queue` for passing all kinds of PyTorch objects between processes. It is possible to e.g. inherit the tensors and storages already in shared memory, when using the `fork` start method, however it is very bug prone and should be used with care, and only by advanced users. Queues, even though they're sometimes a less elegant solution, will work properly in all cases.

: You should be careful about having global statements, that are not guarded with an `if __name__ == '__main__'`. If a different start method than `fork` is used, they will be executed in all subprocesses.

Hogwild

A concrete Hogwild implementation can be found in the examples repository, but to showcase the overall structure of the code, there's also a minimal example below as well:

```
import torch.multiprocessing as mp
from model import MyModel

def train(model):
    # Construct data_loader, optimizer, etc.
    for data, labels in data_loader:
        optimizer.zero_grad()
        loss_fn(model(data), labels).backward()
        optimizer.step()  # This will update the shared parameters
```

```
if __name__ == '__main__':
    num_processes = 4
    model = MyModel()
    # NOTE: this is required for the ``fork`` method to work
    model.share_memory()
    processes = []
    for rank in range(num_processes):
        p = mp.Process(target=train, args=(model,))
        p.start()
        processes.append(p)
    for p in processes:
      p.join()
```

CHAPTER 5

Serialization semantics

Best practices

Recommended approach for saving a model

There are two main approaches for serializing and restoring a model.

The first (recommended) saves and loads only the model parameters:

```
torch.save(the_model.state_dict(), PATH)
```

Then later:

```
the_model = TheModelClass(*args, **kwargs)
the_model.load_state_dict(torch.load(PATH))
```

The second saves and loads the entire model:

```
torch.save(the_model, PATH)
```

Then later:

```
the_model = torch.load(PATH)
```

However in this case, the serialized data is bound to the specific classes and the exact directory structure used, so it can break in various ways when used in other projects, or after some serious refactors.

CHAPTER 6

torch

The torch package contains data structures for multi-dimensional tensors and mathematical operations over these are defined. Additionally, it provides many utilities for efficient serializing of Tensors and arbitrary types, and other useful utilities.

It has a CUDA counterpart, that enables you to run your tensor computations on an NVIDIA GPU with compute capability >= 2.0.

Tensors

`torch.`**`is_tensor`**(*obj*)
Returns True if *obj* is a pytorch tensor.

obj (*Object*) – Object to test

`torch.`**`is_storage`**(*obj*)
Returns True if *obj* is a pytorch storage object.

obj (*Object*) – Object to test

`torch.`**`set_default_tensor_type`**(*t*)

`torch.`**`numel`**(*input*) → int
Returns the total number of elements in the `input` Tensor.

input (`Tensor`) – the input *Tensor*

Example:

```
>>> a = torch.randn(1,2,3,4,5)
>>> torch.numel(a)
120
>>> a = torch.zeros(4,4)
>>> torch.numel(a)
16
```

torch.**set_printoptions**(*precision=None*, *threshold=None*, *edgeitems=None*, *linewidth=None*, *profile=None*)

Set options for printing. Items shamelessly taken from Numpy

- **precision** – Number of digits of precision for floating point output (default 8).
- **threshold** – Total number of array elements which trigger summarization rather than full repr (default 1000).
- **edgeitems** – Number of array items in summary at beginning and end of each dimension (default 3).
- **linewidth** – The number of characters per line for the purpose of inserting line breaks (default 80). Thresholded matricies will ignore this parameter.
- **profile** – Sane defaults for pretty printing. Can override with any of the above options. (default, short, full)

Creation Ops

torch.**eye**(*n*, *m=None*, *out=None*)

Returns a 2-D tensor with ones on the diagonal and zeros elsewhere.

- **n** (*int*) – Number of rows
- **m** (*int, optional*) – Number of columns. If None, defaults to *n*
- **out** (Tensor, *optional*) – Output tensor

a 2-D tensor with ones on the diagonal and zeros elsewhere

Tensor

Example:

```
>>> torch.eye(3)
 1  0  0
 0  1  0
 0  0  1
[torch.FloatTensor of size 3x3]
```

torch.**from_numpy**(*ndarray*) → Tensor

Creates a *Tensor* from a numpy.ndarray.

The returned tensor and *ndarray* share the same memory. Modifications to the tensor will be reflected in the *ndarray* and vice versa. The returned tensor is not resizable.

Example:

```
>>> a = numpy.array([1, 2, 3])
>>> t = torch.from_numpy(a)
>>> t
torch.LongTensor([1, 2, 3])
>>> t[0] = -1
>>> a
array([-1,  2,  3])
```

torch.**linspace**(*start*, *end*, *steps=100*, *out=None*) → Tensor

Returns a one-dimensional Tensor of `steps` equally spaced points between `start` and `end`

The output tensor is 1D of size `steps`

- **start** (*float*) – The starting value for the set of points
- **end** (*float*) – The ending value for the set of points
- **steps** (*int*) – Number of points to sample between `start` and `end`
- **out** (Tensor, *optional*) – The result *Tensor*

Example:

```
>>> torch.linspace(3, 10, steps=5)

  3.0000
  4.7500
  6.5000
  8.2500
 10.0000
[torch.FloatTensor of size 5]

>>> torch.linspace(-10, 10, steps=5)

-10
 -5
  0
  5
 10
[torch.FloatTensor of size 5]

>>> torch.linspace(start=-10, end=10, steps=5)

-10
 -5
  0
  5
 10
[torch.FloatTensor of size 5]
```

torch.**logspace**(*start*, *end*, *steps=100*, *out=None*) → Tensor

Returns a one-dimensional Tensor of `steps` points logarithmically spaced between 10^{start} and 10^{end}

The output is a 1D tensor of size `steps`

- **start** (*float*) – The starting value for the set of points
- **end** (*float*) – The ending value for the set of points
- **steps** (*int*) – Number of points to sample between `start` and `end`
- **out** (Tensor, *optional*) – The result *Tensor*

Example:

```
>>> torch.logspace(start=-10, end=10, steps=5)
```

```
 1.0000e-10
 1.0000e-05
 1.0000e+00
 1.0000e+05
 1.0000e+10
[torch.FloatTensor of size 5]

>>> torch.logspace(start=0.1, end=1.0, steps=5)

  1.2589
  2.1135
  3.5481
  5.9566
 10.0000
[torch.FloatTensor of size 5]
```

torch.**ones**(**sizes*, *out=None*) → Tensor

Returns a Tensor filled with the scalar value *1*, with the shape defined by the varargs `sizes`.

- **sizes** (*int...*) – a set of ints defining the shape of the output Tensor.
- **out** (Tensor, *optional*) – the result Tensor

Example:

```
>>> torch.ones(2, 3)

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

>>> torch.ones(5)

 1
 1
 1
 1
 1
[torch.FloatTensor of size 5]
```

torch.**rand**(**sizes*, *out=None*) → Tensor

Returns a Tensor filled with random numbers from a uniform distribution on the interval $[0, 1)$

The shape of the Tensor is defined by the varargs `sizes`.

- **sizes** (*int...*) – a set of ints defining the shape of the output Tensor.
- **out** (Tensor, *optional*) – the result Tensor

Example:

```
>>> torch.rand(4)

 0.9193
 0.3347
 0.3232
 0.7715
```

```
[torch.FloatTensor of size 4]

>>> torch.rand(2, 3)

 0.5010  0.5140  0.0719
 0.1435  0.5636  0.0538
[torch.FloatTensor of size 2x3]
```

`torch.`**`randn`**(**sizes*, *out=None*) → Tensor

Returns a Tensor filled with random numbers from a normal distribution with zero mean and variance of one.

The shape of the Tensor is defined by the varargs `sizes`.

- **`sizes`** (*`int...`*) – a set of ints defining the shape of the output Tensor.
- **`out`** (`Tensor, optional`) – the result Tensor

Example:

```
>>> torch.randn(4)

-0.1145
 0.0094
-1.1717
 0.9846
[torch.FloatTensor of size 4]

>>> torch.randn(2, 3)

 1.4339  0.3351 -1.0999
 1.5458 -0.9643 -0.3558
[torch.FloatTensor of size 2x3]
```

`torch.`**`randperm`**(*n*, *out=None*) → LongTensor

Returns a random permutation of integers from `0` to `n - 1`.

`n` (*`int`*) – the upper bound (exclusive)

Example:

```
>>> torch.randperm(4)

 2
 1
 3
 0
[torch.LongTensor of size 4]
```

`torch.`**`range`**(*start*, *end*, *step=1*, *out=None*) → Tensor

returns a 1D Tensor of size $floor((end - start)/step) + 1$ with values from `start` to `end` with step `step`. Step is the gap between two values in the tensor. $x_{i+1} = x_i + step$

- **`start`** (*`float`*) – The starting value for the set of points
- **`end`** (*`float`*) – The ending value for the set of points
- **`step`** (*`float`*) – The gap between each pair of adjacent points

- **out** (Tensor, *optional*) – The result *Tensor*

Example:

```
>>> torch.range(1, 4)

 1
 2
 3
 4
[torch.FloatTensor of size 4]

>>> torch.range(1, 4, 0.5)

 1.0000
 1.5000
 2.0000
 2.5000
 3.0000
 3.5000
 4.0000
[torch.FloatTensor of size 7]
```

torch.**zeros**(**sizes*, *out=None*) → Tensor

Returns a Tensor filled with the scalar value *0*, with the shape defined by the varargs `sizes`.

- **sizes** (*int...*) – a set of ints defining the shape of the output Tensor.
- **out** (Tensor, *optional*) – the result Tensor

Example:

```
>>> torch.zeros(2, 3)

 0  0  0
 0  0  0
[torch.FloatTensor of size 2x3]

>>> torch.zeros(5)

 0
 0
 0
 0
 0
[torch.FloatTensor of size 5]
```

Indexing, Slicing, Joining, Mutating Ops

torch.**cat**(*inputs*, *dimension=0*) → Tensor

Concatenates the given sequence of `inputs` Tensors in the given dimension.

torch.cat() can be seen as an inverse operation for *torch.split()* and *torch.chunk()*

cat() can be best understood via examples.

- **inputs** (*sequence of Tensors*) – Can be any python sequence of *Tensor* of the same type.
- **dimension** (*int, optional*) – The dimension over which the tensors are concatenated

Example:

```
>>> x = torch.randn(2, 3)
>>> x

 0.5983 -0.0341  2.4918
 1.5981 -0.5265 -0.8735
[torch.FloatTensor of size 2x3]

>>> torch.cat((x, x, x), 0)

 0.5983 -0.0341  2.4918
 1.5981 -0.5265 -0.8735
 0.5983 -0.0341  2.4918
 1.5981 -0.5265 -0.8735
 0.5983 -0.0341  2.4918
 1.5981 -0.5265 -0.8735
[torch.FloatTensor of size 6x3]

>>> torch.cat((x, x, x), 1)

 0.5983 -0.0341  2.4918  0.5983 -0.0341  2.4918  0.5983 -0.0341  2.4918
 1.5981 -0.5265 -0.8735  1.5981 -0.5265 -0.8735  1.5981 -0.5265 -0.8735
[torch.FloatTensor of size 2x9]
```

torch.**chunk**(*tensor*, *chunks*, *dim=0*)

Splits a tensor into a number of chunks along a given dimension.

- **tensor** (Tensor) – tensor to split.
- **chunks** (*int*) – number of chunks to return.
- **dim** (*int*) – dimension along which to split the tensor.

torch.**gather**(*input*, *dim*, *index*, *out=None*) → Tensor

Gathers values along an axis specified by *dim*.

For a 3-D tensor the output is specified by:

```
out[i][j][k] = tensor[index[i][j][k]][j][k]  # dim=0
out[i][j][k] = tensor[i][index[i][j][k]][k]  # dim=1
out[i][j][k] = tensor[i][j][index[i][j][k]]  # dim=3
```

- **input** (Tensor) – The source tensor
- **dim** (*int*) – The axis along which to index
- **index** (*LongTensor*) – The indices of elements to gather
- **out** (Tensor, *optional*) – Destination tensor

Example:

```
>>> t = torch.Tensor([[1,2],[3,4]])
>>> torch.gather(t, 1, torch.LongTensor([[0,0],[1,0]]))
 1  1
 4  3
[torch.FloatTensor of size 2x2]
```

torch.**index_select**(*input*, *dim*, *index*, *out=None*) → Tensor

Returns a new *Tensor* which indexes the `input` *Tensor* along dimension `dim` using the entries in `index` which is a *LongTensor*.

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*

- **input** (Tensor) – Input data
- **dim** (*int*) – the dimension in which we index
- **index** (*LongTensor*) – the 1D tensor containing the indices to index
- **out** (Tensor, *optional*) – Output argument

Example:

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

 1.2045  2.4084  0.4001  1.1372
 0.5596  1.5677  0.6219 -0.7954
 1.3635 -1.2313 -0.5414 -1.8478
[torch.FloatTensor of size 3x4]

>>> indices = torch.LongTensor([0, 2])
>>> torch.index_select(x, 0, indices)

 1.2045  2.4084  0.4001  1.1372
 1.3635 -1.2313 -0.5414 -1.8478
[torch.FloatTensor of size 2x4]

>>> torch.index_select(x, 1, indices)

 1.2045  0.4001
 0.5596  0.6219
 1.3635 -0.5414
[torch.FloatTensor of size 3x2]
```

torch.**masked_select**(*input*, *mask*, *out=None*) → Tensor

Returns a new 1D *Tensor* which indexes the `input` *Tensor* according to the binary mask `mask` which is a *ByteTensor*.

The `mask` tensor needs to have the same number of elements as `input`, but it's shape or dimensionality are irrelevant.

: The returned *Tensor* does **not** use the same storage as the original *Tensor*

- **input** (Tensor) – Input data
- **mask** (*ByteTensor*) – the tensor containing the binary mask to index with
- **out** (Tensor, *optional*) – Output argument

Example:

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

 1.2045  2.4084  0.4001  1.1372
 0.5596  1.5677  0.6219 -0.7954
 1.3635 -1.2313 -0.5414 -1.8478
[torch.FloatTensor of size 3x4]

>>> mask = x.ge(0.5)
>>> mask

 1  1  0  1
 1  1  1  0
 1  0  0  0
[torch.ByteTensor of size 3x4]

>>> torch.masked_select(x, mask)

 1.2045
 2.4084
 1.1372
 0.5596
 1.5677
 0.6219
 1.3635
[torch.FloatTensor of size 7]
```

torch.**nonzero**(*input*, *out=None*) → LongTensor

Returns a tensor containing the indices of all non-zero elements of input. Each row in the result contains the indices of a non-zero element in input.

If input has *n* dimensions, then the resulting indices Tensor out is of size *z x n*, where *z* is the total number of non-zero elements in the input Tensor.

- **input** (Tensor) – the input *Tensor*
- **out** (*LongTensor*, *optional*) – The result *Tensor* containing indices

Example:

```
>>> torch.nonzero(torch.Tensor([1, 1, 1, 0, 1]))

 0
 1
 2
 4
[torch.LongTensor of size 4x1]

>>> torch.nonzero(torch.Tensor([[0.6, 0.0, 0.0, 0.0],
```

```
...                                [0.0, 0.4, 0.0, 0.0],
...                                [0.0, 0.0, 1.2, 0.0],
...                                [0.0, 0.0, 0.0,-0.4]]))

 0  0
 1  1
 2  2
 3  3
[torch.LongTensor of size 4x2]
```

torch.**split**(*tensor*, *split_size*, *dim=0*)

Splits the tensor into equally sized chunks (if possible).

Last chunk will be smaller if the tensor size along a given dimension is not divisible by `split_size`.

- **tensor** (Tensor) – tensor to split.
- **split_size** (*int*) – size of a single chunk.
- **dim** (*int*) – dimension along which to split the tensor.

torch.**squeeze**(*input*, *dim=None*, *out=None*)

Returns a *Tensor* with all the dimensions of `input` of size *1* removed.

If *input* is of shape: $(Ax1xBxCx1xD)$ then the *out* Tensor will be of shape: $(AxBxCxD)$

When `dim` is given, a squeeze operation is done only in the given dimension. If *input* is of shape: $(Ax1xB)$, *squeeze(input, 0)* leaves the Tensor unchanged, but *squeeze(input, 1)* will squeeze the tensor to the shape (AxB).

: The returned Tensor shares the storage with the input Tensor, so changing the contents of one will change the contents of the other.

- **input** (Tensor) – the input *Tensor*
- **dim** (*int, optional*) – if given, the input will be squeezed only in this dimension
- **out** (Tensor, *optional*) – The result *Tensor*

Example:

```
>>> x = torch.zeros(2,1,2,1,2)
>>> x.size()
(2L, 1L, 2L, 1L, 2L)
>>> y = torch.squeeze(x)
>>> y.size()
(2L, 2L, 2L)
>>> y = torch.squeeze(x, 0)
>>> y.size()
(2L, 1L, 2L, 1L, 2L)
>>> y = torch.squeeze(x, 1)
>>> y.size()
(2L, 2L, 1L, 2L)
```

torch.**stack**(*sequence*, *dim=0*)

Concatenates sequence of tensors along a new dimension.

All tensors need to be of the same size.

- **sqequence** (*Sequence*) – sequence of tensors to concatenate.
- **dim** (*int*) – dimension to insert. Has to be between 0 and the number of dimensions of concatenated tensors (inclusive).

torch.**t**(*input*, *out=None*) → Tensor

Expects `input` to be a matrix (2D Tensor) and transposes dimensions 0 and 1.

Can be seen as a short-hand function for *transpose(input, 0, 1)*

- **input** (Tensor) – the input *Tensor*
- **out** (Tensor, *optional*) – The result *Tensor*

Example:

```
>>> x = torch.randn(2, 3)
>>> x

 0.4834  0.6907  1.3417
-0.1300  0.5295  0.2321
[torch.FloatTensor of size 2x3]

>>> torch.t(x)

 0.4834 -0.1300
 0.6907  0.5295
 1.3417  0.2321
[torch.FloatTensor of size 3x2]
```

torch.**transpose**(*input*, *dim0*, *dim1*, *out=None*) → Tensor

Returns a *Tensor* that is a transposed version of `input`. The given dimensions `dim0` and `dim1` are swapped.

The resulting `out` Tensor shares it's underlying storage with the `input` Tensor, so changing the content of one would change the content of the other.

- **input** (Tensor) – the input *Tensor*
- **dim0** (*int*) – The first dimension to be transposed
- **dim1** (*int*) – The second dimension to be transposed

Example:

```
>>> x = torch.randn(2, 3)
>>> x

 0.5983 -0.0341  2.4918
 1.5981 -0.5265 -0.8735
[torch.FloatTensor of size 2x3]

>>> torch.transpose(x, 0, 1)

 0.5983  1.5981
-0.0341 -0.5265
```

```
 2.4918 -0.8735
[torch.FloatTensor of size 3x2]
```

torch.**unbind**(*tensor*, *dim=0*)

Removes a tensor dimension.

Returns a tuple of all slices along a given dimension, already without it.

- **tensor** (Tensor) – tensor to unbind.
- **dim** (*int*) – dimension to remove.

torch.**unsqueeze**(*input*, *dim*, *out=None*)

Returns a new tensor with a dimension of size one inserted at the specified position.

The returned tensor shares the same underlying data with this tensor.

- **input** (Tensor) – the input *Tensor*
- **dim** (*int*) – The index at which to insert the singleton dimension
- **out** (Tensor, *optional*) – The result *Tensor*

Example

```
>>> x = torch.Tensor([1, 2, 3, 4])
>>> torch.unsqueeze(x, 0)
 1  2  3  4
[torch.FloatTensor of size 1x4]
>>> torch.unsqueeze(x, 1)
 1
 2
 3
 4
[torch.FloatTensor of size 4x1]
```

Random sampling

torch.**manual_seed**(*seed*)

Sets the seed for generating random numbers. And returns a *torch._C.Generator* object.

seed (*int or long*) – The desired seed.

torch.**initial_seed**()

Returns the initial seed for generating random numbers as a python *long*.

torch.**get_rng_state**()

Returns the random number generator state as a ByteTensor.

torch.**set_rng_state**(*new_state*)

Sets the random number generator state.

new_state (*torch.ByteTensor*) – The desired state

torch.**default_generator** = **<torch._C.Generator object>**

torch.**bernoulli**(*input*, *out=None*) → Tensor

Draws binary random numbers (0 or 1) from a bernoulli distribution.

The `input` Tensor should be a tensor containing probabilities to be used for drawing the binary random number. Hence, all values in `input` have to be in the range: $0 <= input_i <= 1$

The *i-th* element of the output tensor will draw a value *1* according to the *i-th* probability value given in `input`.

The returned `out` Tensor only has values 0 or 1 and is of the same shape as `input`

- **input** (Tensor) – Probability values for the bernoulli distribution
- **out** (Tensor, *optional*) – Output tensor

Example:

```
>>> a = torch.Tensor(3, 3).uniform_(0, 1) # generate a uniform random matrix with
↪range [0, 1]
>>> a

 0.7544  0.8140  0.9842
 0.5282  0.0595  0.6445
 0.1925  0.9553  0.9732
[torch.FloatTensor of size 3x3]

>>> torch.bernoulli(a)

 1  1  1
 0  0  1
 0  1  1
[torch.FloatTensor of size 3x3]

>>> a = torch.ones(3, 3) # probability of drawing "1" is 1
>>> torch.bernoulli(a)

 1  1  1
 1  1  1
 1  1  1
[torch.FloatTensor of size 3x3]

>>> a = torch.zeros(3, 3) # probability of drawing "1" is 0
>>> torch.bernoulli(a)

 0  0  0
 0  0  0
 0  0  0
[torch.FloatTensor of size 3x3]
```

torch.**multinomial**(*input*, *num_samples*, *replacement=False*, *out=None*) → LongTensor

Returns a Tensor where each row contains `num_samples` indices sampled from the multinomial probability distribution located in the corresponding row of Tensor `input`.

: The rows of `input` 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 `input` is a vector, `out` is a vector of size *num_samples*.

If `input` is a matrix with *m* rows, `out` is an matrix of shape $m \times n$.

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 `num_samples` must be lower than `input` length (or number of columns of `input` if it is a matrix).

- **input** (`Tensor`) – Tensor containing probabilities
- **num_samples** (*int*) – number of samples to draw
- **replacement** (*bool, optional*) – Whether to draw with replacement or not
- **out** (`Tensor`, *optional*) – The result *Tensor*

Example:

```
>>> weights = torch.Tensor([0, 10, 3, 0]) # create a Tensor of weights
>>> torch.multinomial(weights, 4)

 1
 2
 0
 0
[torch.LongTensor of size 4]

>>> torch.multinomial(weights, 4, replacement=True)

 1
 2
 1
 2
[torch.LongTensor of size 4]
```

`torch.normal()`

`torch.normal`(*means*, *std*, *out=None*)

Returns a Tensor of random numbers drawn from separate normal distributions who's mean and standard deviation are given.

The `means` is a Tensor with the mean of each output element's normal distribution

The *std* is a Tensor with the standard deviation of each output element's normal distribution

The shapes of `means` and *std* don't need to match. The total number of elements in each Tensor need to be the same.

: When the shapes do not match, the shape of `means` is used as the shape for the returned output Tensor

- **means** (`Tensor`) – the Tensor of per-element means

- **std** (Tensor) – the Tensor of per-element standard deviations
- **out** (Tensor) – the optional result Tensor

Example:

```
torch.normal(means=torch.range(1, 10), std=torch.range(1, 0.1, -0.1))

 1.5104
 1.6955
 2.4895
 4.9185
 4.9895
 6.9155
 7.3683
 8.1836
 8.7164
 9.8916
[torch.FloatTensor of size 10]
```

torch.**normal**(*mean=0.0*, *std*, *out=None*)

Similar to the function above, but the means are shared among all drawn elements.

- **means** (*float, optional*) – the mean for all distributions
- **std** (Tensor) – the Tensor of per-element standard deviations
- **out** (Tensor) – the optional result Tensor

Example:

```
>>> torch.normal(mean=0.5, std=torch.range(1, 5))

  0.5723
  0.0871
 -0.3783
 -2.5689
 10.7893
[torch.FloatTensor of size 5]
```

torch.**normal**(*means*, *std=1.0*, *out=None*)

Similar to the function above, but the standard-deviations are shared among all drawn elements.

- **means** (Tensor) – the Tensor of per-element means
- **std** (*float, optional*) – the standard deviation for all distributions
- **out** (Tensor) – the optional result Tensor

Example:

```
>>> torch.normal(means=torch.range(1, 5))

 1.1681
 2.8884
 3.7718
 2.5616
```

```
 4.2500
[torch.FloatTensor of size 5]
```

Serialization

torch.**save**(*obj*, *f*, *pickle_module=<module 'cPickle' (built-in)>*, *pickle_protocol=2*)
Saves an object to a disk file.

See also: *Recommended approach for saving a model*

- **obj** – saved object
- **f** – a file-like object (has to implement fileno that returns a file descriptor) or a string containing a file name
- **pickle_module** – module used for pickling metadata and objects
- **pickle_protocol** – can be specified to override the default protocol

torch.**load**(*f*, *map_location=None*, *pickle_module=<module 'cPickle' (built-in)>*)
Loads an object saved with *torch.save()* from a file.

torch.load can dynamically remap storages to be loaded on a different device using the map_location argument. If it's a callable, it will be called with two arguments: storage and location tag. It's expected to either return a storage that's been moved to a different location, or None (and the location will be resolved using the default method). If this argument is a dict it's expected to be a mapping from location tags used in a file, to location tags of the current system.

By default the location tags are 'cpu' for host tensors and 'cuda:device_id' (e.g. 'cuda:2') for cuda tensors. User extensions can register their own tagging and deserialization methods using register_package.

- **f** – a file-like object (has to implement fileno that returns a file descriptor, and must implement seek), or a string containing a file name
- **map_location** – a function or a dict specifying how to remap storage locations
- **pickle_module** – module used for unpickling metadata and objects (has to match the pickle_module used to serialize file)

Example

```
>>> torch.load('tensors.pt')
# Load all tensors onto the CPU
>>> torch.load('tensors.pt', map_location=lambda storage, loc: storage)
# Map tensors from GPU 1 to GPU 0
>>> torch.load('tensors.pt', map_location={'cuda:1':'cuda:0'})
```

Parallelism

torch.**get_num_threads**() → int
Gets the number of OpenMP threads used for parallelizing CPU operations

torch.**set_num_threads**(*int*)
Sets the number of OpenMP threads used for parallelizing CPU operations

Math operations

Pointwise Ops

torch.**abs**(*input*, *out=None*) → Tensor
Computes the element-wise absolute value of the given `input` a tensor.

Example:

```
>>> torch.abs(torch.FloatTensor([-1, -2, 3]))
FloatTensor([1, 2, 3])
```

torch.**acos**(*input*, *out=None*) → Tensor
Returns a new *Tensor* with the arccosine of the elements of `input`.

- **input** (Tensor) – the input *Tensor*
- **out** (Tensor, *optional*) – The result *Tensor*

Example:

```
>>> a = torch.randn(4)
>>> a

-0.6366
 0.2718
 0.4469
 1.3122
[torch.FloatTensor of size 4]

>>> torch.acos(a)
 2.2608
 1.2956
 1.1075
    nan
[torch.FloatTensor of size 4]
```

torch.**add**()

torch.**add**(*input*, *value*, *out=None*)

Adds the scalar `value` to each element of the input `input` and returns a new resulting tensor.

$out = tensor + value$

If `input` is of type FloatTensor or DoubleTensor, `value` must be a real number, otherwise it should be an integer

- **input** (Tensor) – the input *Tensor*
- **value** (*Number*) – the number to be added to each element of `input`

- **out** (Tensor, *optional*) – The result *Tensor*

Example:

```
>>> a = torch.randn(4)
>>> a

 0.4050
-1.2227
 1.8688
-0.4185
[torch.FloatTensor of size 4]

>>> torch.add(a, 20)

 20.4050
 18.7773
 21.8688
 19.5815
[torch.FloatTensor of size 4]
```

torch.**add**(*input*, *value=1*, *other*, *out=None*)

Each element of the Tensor `other` is multiplied by the scalar `value` and added to each element of the Tensor `input`. The resulting Tensor is returned.

The shapes of `input` and `other` don't need to match. The total number of elements in each Tensor need to be the same.

: When the shapes do not match, the shape of `input` is used as the shape for the returned output Tensor

---

$$out = input + (other * value)$$

If `other` is of type FloatTensor or DoubleTensor, `value` must be a real number, otherwise it should be an integer

- **input** (Tensor) – the first input *Tensor*
- **value** (*Number*) – the scalar multiplier for `other`
- **other** (Tensor) – the second input *Tensor*
- **out** (Tensor, *optional*) – The result *Tensor*

Example:

```
>>> import torch
>>> a = torch.randn(4)
>>> a

-0.9310
 2.0330
 0.0852
-0.2941
[torch.FloatTensor of size 4]

>>> b = torch.randn(2, 2)
>>> b
```

```
 1.0663  0.2544
-0.1513  0.0749
[torch.FloatTensor of size 2x2]

>>> torch.add(a, 10, b)
 9.7322
 4.5770
-1.4279
 0.4552
[torch.FloatTensor of size 4]
```

torch.**addcdiv**(*tensor*, *value=1*, *tensor1*, *tensor2*, *out=None*) → Tensor

Performs the element-wise division of `tensor1` by `tensor2`, multiply the result by the scalar `value` and add it to `tensor`.

The number of elements must match, but sizes do not matter.

For inputs of type *FloatTensor* or *DoubleTensor*, `value` must be a real number, otherwise an integer

- **tensor** (Tensor) – the tensor to be added
- **value** (*Number, optional*) – multiplier for *tensor1 ./ tensor2*
- **tensor1** (Tensor) – Numerator tensor
- **tensor2** (Tensor) – Denominator tensor
- **out** (Tensor, *optional*) – Output tensor

Example:

```
>>> t = torch.randn(2, 3)
>>> t1 = torch.randn(1, 6)
>>> t2 = torch.randn(6, 1)
>>> torch.addcdiv(t, 0.1, t1, t2)

 0.0122 -0.0188 -0.2354
 0.7396 -1.5721  1.2878
[torch.FloatTensor of size 2x3]
```

torch.**addcmul**(*tensor*, *value=1*, *tensor1*, *tensor2*, *out=None*) → Tensor

Performs the element-wise multiplication of `tensor1` by `tensor2`, multiply the result by the scalar `value` and add it to `tensor`.

The number of elements must match, but sizes do not matter.

For inputs of type *FloatTensor* or *DoubleTensor*, `value` must be a real number, otherwise an integer

- **tensor** (Tensor) – the tensor to be added
- **value** (*Number, optional*) – multiplier for *tensor1 .* tensor2*
- **tensor1** (Tensor) – tensor to be multiplied
- **tensor2** (Tensor) – tensor to be multiplied
- **out** (Tensor, *optional*) – Output tensor

Example:

```
>>> t = torch.randn(2, 3)
>>> t1 = torch.randn(1, 6)
>>> t2 = torch.randn(6, 1)
>>> torch.addcmul(t, 0.1, t1, t2)

 0.0122 -0.0188 -0.2354
 0.7396 -1.5721  1.2878
[torch.FloatTensor of size 2x3]
```

`torch.`**`asin`**(*input*, *out=None*) → Tensor

Returns a new *Tensor* with the arcsine of the elements of `input`.

- **input** (Tensor) – the input *Tensor*
- **out** (Tensor, *optional*) – The result *Tensor*

Example:

```
>>> a = torch.randn(4)
>>> a
-0.6366
 0.2718
 0.4469
 1.3122
[torch.FloatTensor of size 4]

>>> torch.asin(a)
-0.6900
 0.2752
 0.4633
    nan
[torch.FloatTensor of size 4]
```

`torch.`**`atan`**(*input*, *out=None*) → Tensor

Returns a new *Tensor* with the arctangent of the elements of `input`.

- **input** (Tensor) – the input *Tensor*
- **out** (Tensor, *optional*) – The result *Tensor*

Example:

```
>>> a = torch.randn(4)
>>> a
-0.6366
 0.2718
 0.4469
 1.3122
[torch.FloatTensor of size 4]

>>> torch.atan(a)
-0.5669
 0.2653
 0.4203
 0.9196
[torch.FloatTensor of size 4]
```

torch.**atan2**(*input1*, *input2*, *out=None*) → Tensor

Returns a new *Tensor* with the arctangent of the elements of `input1` and `input2`.

- **input1** (Tensor) – the first input *Tensor*
- **input2** (Tensor) – the second input *Tensor*
- **out** (Tensor, *optional*) – The result *Tensor*

Example:

```
>>> a = torch.randn(4)
>>> a
-0.6366
 0.2718
 0.4469
 1.3122
[torch.FloatTensor of size 4]

>>> torch.atan2(a, torch.randn(4))
-2.4167
 2.9755
 0.9363
 1.6613
[torch.FloatTensor of size 4]
```

torch.**ceil**(*input*, *out=None*) → Tensor

Returns a new *Tensor* with the ceil of the elements of `input`, the smallest integer greater than or equal to each element.

- **input** (Tensor) – the input *Tensor*
- **out** (Tensor, *optional*) – The result *Tensor*

Example:

```
>>> a = torch.randn(4)
>>> a

 1.3869
 0.3912
-0.8634
-0.5468
[torch.FloatTensor of size 4]

>>> torch.ceil(a)

 2
 1
-0
-0
[torch.FloatTensor of size 4]
```

torch.**clamp**(*input*, *min*, *max*, *out=None*) → Tensor

Clamp all elements in `input` into the range *[min, max]* and return a resulting Tensor.

```
      | min, if x_i < min
y_i = | x_i, if min <= x_i <= max
      | max, if x_i > max
```

If `input` is of type *FloatTensor* or *DoubleTensor*, args *min* and *max* must be real numbers, otherwise they should be integers

- **input** (Tensor) – the input *Tensor*
- **min** (*Number*) – lower-bound of the range to be clamped to
- **max** (*Number*) – upper-bound of the range to be clamped to
- **out** (Tensor, *optional*) – The result *Tensor*

Example:

```
>>> a = torch.randn(4)
>>> a

 1.3869
 0.3912
-0.8634
-0.5468
[torch.FloatTensor of size 4]

>>> torch.clamp(a, min=-0.5, max=0.5)

 0.5000
 0.3912
-0.5000
-0.5000
[torch.FloatTensor of size 4]
```

torch.**clamp**(*input*, *, *min*, *out=None*) → Tensor

Clamps all elements in `input` to be larger or equal *min*.

If `input` is of type *FloatTensor* or *DoubleTensor*, `value` should be a real number, otherwise it should be an integer

- **input** (Tensor) – the input *Tensor*
- **value** (*Number*) – minimal value of each element in the output
- **out** (Tensor, *optional*) – The result *Tensor*

Example:

```
>>> a = torch.randn(4)
>>> a

 1.3869
 0.3912
-0.8634
-0.5468
[torch.FloatTensor of size 4]
```

```
>>> torch.clamp(a, min=0.5)

 1.3869
 0.5000
 0.5000
 0.5000
[torch.FloatTensor of size 4]
```

torch.**clamp**(*input*, *, *max*, *out=None*) → Tensor

Clamps all elements in `input` to be smaller or equal *max*.

If `input` is of type *FloatTensor* or *DoubleTensor*, `value` should be a real number, otherwise it should be an integer

- **input** (Tensor) – the input *Tensor*
- **value** (*Number*) – maximal value of each element in the output
- **out** (Tensor, *optional*) – The result *Tensor*

Example:

```
>>> a = torch.randn(4)
>>> a

 1.3869
 0.3912
-0.8634
-0.5468
[torch.FloatTensor of size 4]

>>> torch.clamp(a, max=0.5)

 0.5000
 0.3912
-0.8634
-0.5468
[torch.FloatTensor of size 4]
```

torch.**cos**(*input*, *out=None*) → Tensor

Returns a new *Tensor* with the cosine of the elements of `input`.

- **input** (Tensor) – the input *Tensor*
- **out** (Tensor, *optional*) – The result *Tensor*

Example:

```
>>> a = torch.randn(4)
>>> a
-0.6366
 0.2718
 0.4469
 1.3122
[torch.FloatTensor of size 4]

>>> torch.cos(a)
```

```
 0.8041
 0.9633
 0.9018
 0.2557
[torch.FloatTensor of size 4]
```

`torch.`**`cosh`**(*input*, *out=None*) → Tensor

Returns a new *Tensor* with the hyperbolic cosine of the elements of `input`.

- **`input`** (`Tensor`) – the input *Tensor*
- **`out`** (`Tensor, optional`) – The result *Tensor*

Example:

```
>>> a = torch.randn(4)
>>> a
-0.6366
 0.2718
 0.4469
 1.3122
[torch.FloatTensor of size 4]

>>> torch.cosh(a)
 1.2095
 1.0372
 1.1015
 1.9917
[torch.FloatTensor of size 4]
```

`torch.`**`div`**()

`torch.`**`div`**(*input*, *value*, *out=None*)

Divides each element of the input `input` with the scalar `value` and returns a new resulting tensor.

$out = tensor/value$

If `input` is of type *FloatTensor* or *DoubleTensor*, `value` should be a real number, otherwise it should be an integer

- **`input`** (`Tensor`) – the input *Tensor*
- **`value`** (`Number`) – the number to be divided to each element of `input`
- **`out`** (`Tensor, optional`) – The result *Tensor*

Example:

```
>>> a = torch.randn(5)
>>> a

-0.6147
-1.1237
-0.1604
-0.6853
 0.1063
```

```
[torch.FloatTensor of size 5]

>>> torch.div(a, 0.5)

-1.2294
-2.2474
-0.3208
-1.3706
 0.2126
[torch.FloatTensor of size 5]
```

torch.**div**(*input*, *other*, *out=None*)

Each element of the Tensor `input` is divided by each element of the Tensor `other`. The resulting Tensor is returned. The shapes of `input` and `other` don't need to match. The total number of elements in each Tensor need to be the same.

: When the shapes do not match, the shape of `input` is used as the shape for the returned output Tensor

---

$$out_i = input_i / other_i$$

- **input** (Tensor) – the numerator *Tensor*
- **other** (Tensor) – the denominator *Tensor*
- **out** (Tensor, *optional*) – The result *Tensor*

Example:

```
>>> a = torch.randn(4,4)
>>> a

-0.1810  0.4017  0.2863 -0.1013
 0.6183  2.0696  0.9012 -1.5933
 0.5679  0.4743 -0.0117 -0.1266
-0.1213  0.9629  0.2682  1.5968
[torch.FloatTensor of size 4x4]

>>> b = torch.randn(8, 2)
>>> b

 0.8774  0.7650
 0.8866  1.4805
-0.6490  1.1172
 1.4259 -0.8146
 1.4633 -0.1228
 0.4643 -0.6029
 0.3492  1.5270
 1.6103 -0.6291
[torch.FloatTensor of size 8x2]

>>> torch.div(a, b)

-0.2062  0.5251  0.3229 -0.0684
-0.9528  1.8525  0.6320  1.9559
 0.3881 -3.8625 -0.0253  0.2099
```

```
-0.3473  0.6306  0.1666 -2.5381
[torch.FloatTensor of size 4x4]
```

torch.**exp**(*tensor*, *out=None*) → Tensor

Computes the exponential of each element.

Example:

```
>>> torch.exp(torch.Tensor([0, math.log(2)]))
torch.FloatTensor([1, 2])
```

torch.**floor**(*input*, *out=None*) → Tensor

Returns a new *Tensor* with the floor of the elements of `input`, the largest integer less than or equal to each element.

- **input** (Tensor) – the input *Tensor*
- **out** (Tensor, *optional*) – The result *Tensor*

Example:

```
>>> a = torch.randn(4)
>>> a

 1.3869
 0.3912
-0.8634
-0.5468
[torch.FloatTensor of size 4]

>>> torch.floor(a)

 1
 0
-1
-1
[torch.FloatTensor of size 4]
```

torch.**fmod**(*input*, *divisor*, *out=None*) → Tensor

Computes the element-wise remainder of division.

The dividend and divisor may contain both for integer and floating point numbers. The remainder has the same sign as the dividend *tensor*.

- **input** (Tensor) – The dividend
- **divisor** (Tensor *or* *float*) – The divisor. This may be either a number or a tensor of the same shape as the dividend.
- **out** (Tensor, *optional*) – Output tensor

Example:

```
>>> torch.fmod(torch.Tensor([-3, -2, -1, 1, 2, 3]), 2)
torch.FloatTensor([-1, -0, -1, 1, 0, 1])
>>> torch.fmod(torch.Tensor([1, 2, 3, 4, 5]), 1.5)
torch.FloatTensor([1.0, 0.5, 0.0, 1.0, 0.5])
```

:

torch.remainder(), which computes the element-wise remainder of division equivalently to Python's % operator

torch.**frac**(*tensor*, *out=None*) → Tensor

Computes the fractional portion of each element in *tensor*.

Example:

```
>>> torch.frac(torch.Tensor([1, 2.5, -3.2])
torch.FloatTensor([0, 0.5, -0.2])
```

torch.**lerp**(*start*, *end*, *weight*, *out=None*)

Does a linear interpolation of two tensors `start` and `end` based on a scalar `weight`: and returns the resulting `out` Tensor.

$$out_i = start_i + weight * (end_i - start_i)$$

- **start** (Tensor) – the *Tensor* with the starting points
- **end** (Tensor) – the *Tensor* with the ending points
- **weight** (*float*) – the weight for the interpolation formula
- **out** (Tensor, *optional*) – The result *Tensor*

Example:

```
>>> start = torch.range(1, 4)
>>> end = torch.Tensor(4).fill_(10)
>>> start

 1
 2
 3
 4
[torch.FloatTensor of size 4]

>>> end

 10
 10
 10
 10
[torch.FloatTensor of size 4]

>>> torch.lerp(start, end, 0.5)

 5.5000
 6.0000
 6.5000
 7.0000
[torch.FloatTensor of size 4]
```

torch.**log**(*input*, *out=None*) → Tensor

Returns a new *Tensor* with the natural logarithm of the elements of `input`.

- **input** (Tensor) – the input *Tensor*

- **out** (Tensor, *optional*) – The result *Tensor*

Example:

```
>>> a = torch.randn(5)
>>> a

-0.4183
 0.3722
-0.3091
 0.4149
 0.5857
[torch.FloatTensor of size 5]

>>> torch.log(a)

    nan
-0.9883
    nan
-0.8797
-0.5349
[torch.FloatTensor of size 5]
```

torch.**log1p**(*input*, *out=None*) → Tensor

Returns a new *Tensor* with the natural logarithm of (1 + input).

$y_i = log(x_i + 1)$

: This function is more accurate than *torch.log()* for small values of input

- **input** (Tensor) – the input *Tensor*
- **out** (Tensor, *optional*) – The result *Tensor*

Example:

```
>>> a = torch.randn(5)
>>> a

-0.4183
 0.3722
-0.3091
 0.4149
 0.5857
[torch.FloatTensor of size 5]

>>> torch.log1p(a)

-0.5418
 0.3164
-0.3697
 0.3471
 0.4611
[torch.FloatTensor of size 5]
```

torch.**mul**()

torch.**mul**(*input*, *value*, *out=None*)

Multiplies each element of the input `input` with the scalar `value` and returns a new resulting tensor.

$out = tensor * value$

If `input` is of type *FloatTensor* or *DoubleTensor*, `value` should be a real number, otherwise it should be an integer

- **input** (Tensor) – the input *Tensor*
- **value** (*Number*) – the number to be multiplied to each element of `input`
- **out** (Tensor, *optional*) – The result *Tensor*

Example:

```
>>> a = torch.randn(3)
>>> a

-0.9374
-0.5254
-0.6069
[torch.FloatTensor of size 3]

>>> torch.mul(a, 100)

-93.7411
-52.5374
-60.6908
[torch.FloatTensor of size 3]
```

torch.**mul**(*input*, *other*, *out=None*)

Each element of the Tensor `input` is multiplied by each element of the Tensor `other`. The resulting Tensor is returned. The shapes of `input` and `other` don't need to match. The total number of elements in each Tensor need to be the same.

: When the shapes do not match, the shape of `input` is used as the shape for the returned output Tensor

---

$out_i = input_i * other_i$

- **input** (Tensor) – the first multiplicand *Tensor*
- **other** (Tensor) – the second multiplicand *Tensor*
- **out** (Tensor, *optional*) – The result *Tensor*

Example:

```
>>> a = torch.randn(4,4)
>>> a

-0.7280  0.0598 -1.4327 -0.5825
-0.1427 -0.0690  0.0821 -0.3270
```

```
-0.9241  0.5110  0.4070 -1.1188
-0.8308  0.7426 -0.6240 -1.1582
[torch.FloatTensor of size 4x4]

>>> b = torch.randn(2, 8)
>>> b

 0.0430 -1.0775  0.6015  1.1647 -0.6549  0.0308 -0.1670  1.0742
-1.2593  0.0292 -0.0849  0.4530  1.2404 -0.4659 -0.1840  0.5974
[torch.FloatTensor of size 2x8]

>>> torch.mul(a, b)

-0.0313 -0.0645 -0.8618 -0.6784
 0.0934 -0.0021 -0.0137 -0.3513
 1.1638  0.0149 -0.0346 -0.5068
-1.0304 -0.3460  0.1148 -0.6919
[torch.FloatTensor of size 4x4]
```

torch.**neg**(*input*, *out=None*) → Tensor

Returns a new *Tensor* with the negative of the elements of `input`.

$out = -1 * input$

- **input** (Tensor) – the input *Tensor*
- **out** (Tensor, *optional*) – The result *Tensor*

Example:

```
>>> a = torch.randn(5)
>>> a

-0.4430
 1.1690
-0.8836
-0.4565
 0.2968
[torch.FloatTensor of size 5]

>>> torch.neg(a)

 0.4430
-1.1690
 0.8836
 0.4565
-0.2968
[torch.FloatTensor of size 5]
```

torch.**pow**()

torch.**pow**(*input*, *exponent*, *out=None*)

Takes the power of each element in `input` with `exponent` and returns a Tensor with the result.

`exponent` can be either a single `float` number or a `Tensor` with the same number of elements as `input`.

When `exponent` is a scalar value, the operation applied is:

$out_i = x_i^{exponent}$

When `exponent` is a Tensor, the operation applied is:

$out_i = x_i^{exponent_i}$

- **input** (Tensor) – the input *Tensor*
- **exponent** (*float or* Tensor) – the exponent value
- **out** (Tensor, *optional*) – The result *Tensor*

Example:

```
>>> a = torch.randn(4)
>>> a

-0.5274
-0.8232
-2.1128
 1.7558
[torch.FloatTensor of size 4]

>>> torch.pow(a, 2)

 0.2781
 0.6776
 4.4640
 3.0829
[torch.FloatTensor of size 4]

>>> exp = torch.range(1, 4)
>>> a = torch.range(1, 4)
>>> a

 1
 2
 3
 4
[torch.FloatTensor of size 4]

>>> exp

 1
 2
 3
 4
[torch.FloatTensor of size 4]

>>> torch.pow(a, exp)

   1
   4
  27
 256
[torch.FloatTensor of size 4]
```

`torch.`**pow**(*base*, *input*, *out=None*)

`base` is a scalar `float` value, and `input` is a Tensor. The returned Tensor `out` is of the same shape as `input`

The operation applied is:

$out_i = base^{input_i}$

- **base** (*float*) – the scalar base value for the power operation
- **input** (Tensor) – the exponent *Tensor*
- **out** (Tensor, *optional*) – The result *Tensor*

Example:

```
>>> exp = torch.range(1, 4)
>>> base = 2
>>> torch.pow(base, exp)

  2
  4
  8
 16
[torch.FloatTensor of size 4]
```

torch.**reciprocal**(*input*, *out=None*) → Tensor

Returns a new *Tensor* with the reciprocal of the elements of input, i.e. $1.0/x$

- **input** (Tensor) – the input *Tensor*
- **out** (Tensor, *optional*) – The result *Tensor*

Example:

```
>>> a = torch.randn(4)
>>> a

 1.3869
 0.3912
-0.8634
-0.5468
[torch.FloatTensor of size 4]

>>> torch.reciprocal(a)

 0.7210
 2.5565
-1.1583
-1.8289
[torch.FloatTensor of size 4]
```

torch.**remainder**(*input*, *divisor*, *out=None*) → Tensor

Computes the element-wise remainder of division.

The divisor and dividend may contain both for integer and floating point numbers. The remainder has the same sign as the divisor.

- **input** (Tensor) – The dividend

- **divisor** (*Tensor or float*) – The divisor. This may be either a number or a tensor of the same shape as the dividend.
- **out** (*Tensor, optional*) – Output tensor

Example:

```
>>> torch.remainder(torch.Tensor([-3, -2, -1, 1, 2, 3]), 2)
torch.FloatTensor([1, 0, 1, 1, 0, 1])
>>> torch.remainder(torch.Tensor([1, 2, 3, 4, 5]), 1.5)
torch.FloatTensor([1.0, 0.5, 0.0, 1.0, 0.5])
```

:

torch.fmod(), which computes the element-wise remainder of division equivalently to the C library function `fmod()`

`torch.`**`round`**(*input*, *out=None*) → Tensor

Returns a new *Tensor* with each of the elements of `input` rounded to the closest integer.

- **input** (*Tensor*) – the input *Tensor*
- **out** (*Tensor, optional*) – The result *Tensor*

Example:

```
>>> a = torch.randn(4)
>>> a

 1.2290
 1.3409
-0.5662
-0.0899
[torch.FloatTensor of size 4]

>>> torch.round(a)

 1
 1
-1
-0
[torch.FloatTensor of size 4]
```

`torch.`**`rsqrt`**(*input*, *out=None*) → Tensor

Returns a new *Tensor* with the reciprocal of the square-root of each of the elements of `input`.

- **input** (*Tensor*) – the input *Tensor*
- **out** (*Tensor, optional*) – The result *Tensor*

Example:

```
>>> a = torch.randn(4)
>>> a

 1.2290
 1.3409
-0.5662
-0.0899
```

```
[torch.FloatTensor of size 4]

>>> torch.rsqrt(a)

 0.9020
 0.8636
    nan
    nan
[torch.FloatTensor of size 4]
```

`torch.`**`sigmoid`**(*input*, *out=None*) → Tensor

Returns a new *Tensor* with the sigmoid of the elements of `input`.

- **`input`** (`Tensor`) – the input *Tensor*
- **`out`** (`Tensor, optional`) – The result *Tensor*

Example:

```
>>> a = torch.randn(4)
>>> a

-0.4972
 1.3512
 0.1056
-0.2650
[torch.FloatTensor of size 4]

>>> torch.sigmoid(a)

 0.3782
 0.7943
 0.5264
 0.4341
[torch.FloatTensor of size 4]
```

`torch.`**`sign`**(*input*, *out=None*) → Tensor

Returns a new *Tensor* with the sign of the elements of `input`.

- **`input`** (`Tensor`) – the input *Tensor*
- **`out`** (`Tensor, optional`) – The result *Tensor*

Example:

```
>>> a = torch.randn(4)
>>> a
-0.6366
 0.2718
 0.4469
 1.3122
[torch.FloatTensor of size 4]

>>> torch.sign(a)

-1
 1
```

```
 1
 1
[torch.FloatTensor of size 4]
```

torch.**sin**(*input*, *out=None*) → Tensor

Returns a new *Tensor* with the sine of the elements of `input`.

- **input** (Tensor) – the input *Tensor*
- **out** (Tensor, *optional*) – The result *Tensor*

Example:

```
>>> a = torch.randn(4)
>>> a
-0.6366
 0.2718
 0.4469
 1.3122
[torch.FloatTensor of size 4]

>>> torch.sin(a)
-0.5944
 0.2684
 0.4322
 0.9667
[torch.FloatTensor of size 4]
```

torch.**sinh**(*input*, *out=None*) → Tensor

Returns a new *Tensor* with the hyperbolic sine of the elements of `input`.

- **input** (Tensor) – the input *Tensor*
- **out** (Tensor, *optional*) – The result *Tensor*

Example:

```
>>> a = torch.randn(4)
>>> a
-0.6366
 0.2718
 0.4469
 1.3122
[torch.FloatTensor of size 4]

>>> torch.sinh(a)
-0.6804
 0.2751
 0.4619
 1.7225
[torch.FloatTensor of size 4]
```

torch.**sqrt**(*input*, *out=None*) → Tensor

Returns a new *Tensor* with the square-root of the elements of `input`.

- **input** (Tensor) – the input *Tensor*
- **out** (Tensor, *optional*) – The result *Tensor*

Example:

```
>>> a = torch.randn(4)
>>> a

 1.2290
 1.3409
-0.5662
-0.0899
[torch.FloatTensor of size 4]

>>> torch.sqrt(a)

 1.1086
 1.1580
    nan
    nan
[torch.FloatTensor of size 4]
```

torch.**tan**(*input*, *out=None*) → Tensor

Returns a new *Tensor* with the tangent of the elements of `input`.

- **input** (Tensor) – the input *Tensor*
- **out** (Tensor, *optional*) – The result *Tensor*

Example:

```
>>> a = torch.randn(4)
>>> a
-0.6366
 0.2718
 0.4469
 1.3122
[torch.FloatTensor of size 4]

>>> torch.tan(a)
-0.7392
 0.2786
 0.4792
 3.7801
[torch.FloatTensor of size 4]
```

torch.**tanh**(*input*, *out=None*) → Tensor

Returns a new *Tensor* with the hyperbolic tangent of the elements of `input`.

- **input** (Tensor) – the input *Tensor*
- **out** (Tensor, *optional*) – The result *Tensor*

Example:

```
>>> a = torch.randn(4)
>>> a
```

```
-0.6366
 0.2718
 0.4469
 1.3122
[torch.FloatTensor of size 4]

>>> torch.tanh(a)
-0.5625
 0.2653
 0.4193
 0.8648
[torch.FloatTensor of size 4]
```

`torch.`**`trunc`**(*input*, *out=None*) → Tensor

Returns a new *Tensor* with the truncated integer values of the elements of `input`.

- **`input`** (`Tensor`) – the input *Tensor*
- **`out`** (`Tensor, optional`) – The result *Tensor*

Example:

```
>>> a = torch.randn(4)
>>> a

-0.4972
 1.3512
 0.1056
-0.2650
[torch.FloatTensor of size 4]

>>> torch.trunc(a)

-0
 1
 0
-0
[torch.FloatTensor of size 4]
```

Reduction Ops

`torch.`**`cumprod`**(*input*, *dim*, *out=None*) → Tensor

Returns the cumulative product of elements of `input` in the dimension `dim`.

For example, if `input` is a vector of size N, the result will also be a vector of size N, with elements: $y_i = x_1 * x_2 * x_3 * ... * x_i$

- **`input`** (`Tensor`) – the input *Tensor*
- **`dim`** (`int`) – the dimension to do the operation over
- **`out`** (`Tensor, optional`) – The result *Tensor*

Example:

```
>>> a = torch.randn(10)
>>> a

 1.1148
 1.8423
 1.4143
-0.4403
 1.2859
-1.2514
-0.4748
 1.1735
-1.6332
-0.4272
[torch.FloatTensor of size 10]

>>> torch.cumprod(a, dim=0)

 1.1148
 2.0537
 2.9045
-1.2788
-1.6444
 2.0578
-0.9770
-1.1466
 1.8726
-0.8000
[torch.FloatTensor of size 10]

>>> a[5] = 0.0
>>> torch.cumprod(a, dim=0)

 1.1148
 2.0537
 2.9045
-1.2788
-1.6444
-0.0000
 0.0000
 0.0000
-0.0000
 0.0000
[torch.FloatTensor of size 10]
```

`torch.`**`cumsum`**(*input*, *dim*, *out=None*) → Tensor

Returns the cumulative sum of elements of `input` in the dimension `dim`.

For example, if `input` is a vector of size N, the result will also be a vector of size N, with elements: $y_i = x_1 + x_2 + x_3 + ... + x_i$

- **`input`** (`Tensor`) – the input *Tensor*
- **`dim`** (*`int`*) – the dimension to do the operation over
- **`out`** (`Tensor`, *`optional`*) – The result *Tensor*

Example:

```
>>> a = torch.randn(10)
>>> a

-0.6039
-0.2214
-0.3705
-0.0169
 1.3415
-0.1230
 0.9719
 0.6081
-0.1286
 1.0947
[torch.FloatTensor of size 10]

>>> torch.cumsum(a, dim=0)

-0.6039
-0.8253
-1.1958
-1.2127
 0.1288
 0.0058
 0.9777
 1.5858
 1.4572
 2.5519
[torch.FloatTensor of size 10]
```

`torch.`**`dist`**(*input*, *other*, *p=2*, *out=None*) → Tensor

Returns the p-norm of (`input - other`)

- **`input`** (`Tensor`) – the input *Tensor*
- **`other`** (`Tensor`) – the Right-hand-side input *Tensor*
- **`p`** (*`float, optional`*) – The norm to be computed.
- **`out`** (`Tensor, optional`) – The result *Tensor*

Example:

```
>>> x = torch.randn(4)
>>> x

 0.2505
-0.4571
-0.3733
 0.7807
[torch.FloatTensor of size 4]

>>> y = torch.randn(4)
>>> y

 0.7782
-0.5185
 1.4106
-2.4063
```

```
[torch.FloatTensor of size 4]

>>> torch.dist(x, y, 3.5)
3.302832063224223
>>> torch.dist(x, y, 3)
3.3677282206393286
>>> torch.dist(x, y, 0)
inf
>>> torch.dist(x, y, 1)
5.560028076171875
```

torch.**mean**()

torch.**mean**(*input*) → float

Returns the mean value of all elements in the `input` Tensor.

input (Tensor) – the input *Tensor*

Example:

```
>>> a = torch.randn(1, 3)
>>> a

-0.2946 -0.9143  2.1809
[torch.FloatTensor of size 1x3]

>>> torch.mean(a)
0.32398951053619385
```

torch.**mean**(*input*, *dim*, *out=None*) → Tensor

Returns the mean value of each row of the `input` Tensor in the given dimension `dim`.

The output Tensor is of the same size as `input` except in the dimension `dim` where it is of size 1.

- **input** (Tensor) – the input *Tensor*
- **dim** (*int*) – the dimension to reduce
- **out** (Tensor, *optional*) – the result Tensor

Example:

```
>>> a = torch.randn(4, 4)
>>> a

-1.2738 -0.3058  0.1230 -1.9615
 0.8771 -0.5430 -0.9233  0.9879
 1.4107  0.0317 -0.6823  0.2255
-1.3854  0.4953 -0.2160  0.2435
[torch.FloatTensor of size 4x4]

>>> torch.mean(a, 1)

-0.8545
 0.0997
 0.2464
```

```
-0.2157
[torch.FloatTensor of size 4x1]
```

torch.**median**(*input*, *dim=-1*, *values=None*, *indices=None*) -> (*Tensor*, *LongTensor*)

Returns the median value of each row of the `input` Tensor in the given dimension `dim`. Also returns the index location of the median value as a *LongTensor*.

By default, `dim` is the last dimension of the `input` Tensor.

The output Tensors are of the same size as `input` except in the dimension `dim` where it is of size 1.

: This function is not defined for `torch.cuda.Tensor` yet.

- **input** (Tensor) – the input *Tensor*
- **dim** (*int*) – the dimension to reduce
- **values** (Tensor, *optional*) – the result Tensor
- **indices** (Tensor, *optional*) – the result index Tensor

Example:

```
>>> a

 -0.6891 -0.6662
 0.2697  0.7412
 0.5254 -0.7402
 0.5528 -0.2399
[torch.FloatTensor of size 4x2]

>>> a = torch.randn(4, 5)
>>> a

 0.4056 -0.3372  1.0973 -2.4884  0.4334
 2.1336  0.3841  0.1404 -0.1821 -0.7646
-0.2403  1.3975 -2.0068  0.1298  0.0212
-1.5371 -0.7257 -0.4871 -0.2359 -1.1724
[torch.FloatTensor of size 4x5]

>>> torch.median(a, 1)
(
 0.4056
 0.1404
 0.0212
-0.7257
[torch.FloatTensor of size 4x1]
,
 0
 2
 4
 1
[torch.LongTensor of size 4x1]
)
```

torch.**mode**(*input*, *dim=-1*, *values=None*, *indices=None) -> (Tensor*, *LongTensor*)

Returns the mode value of each row of the `input` Tensor in the given dimension `dim`. Also returns the index location of the mode value as a *LongTensor*.

By default, `dim` is the last dimension of the `input` Tensor.

The output Tensors are of the same size as `input` except in the dimension `dim` where it is of size 1.

: This function is not defined for `torch.cuda.Tensor` yet.

- **input** (Tensor) – the input *Tensor*
- **dim** (*int*) – the dimension to reduce
- **values** (Tensor, *optional*) – the result Tensor
- **indices** (Tensor, *optional*) – the result index Tensor

Example:

```
>>> a

 -0.6891 -0.6662
 0.2697  0.7412
 0.5254 -0.7402
 0.5528 -0.2399
[torch.FloatTensor of size 4x2]

>>> a = torch.randn(4, 5)
>>> a

 0.4056 -0.3372  1.0973 -2.4884  0.4334
 2.1336  0.3841  0.1404 -0.1821 -0.7646
-0.2403  1.3975 -2.0068  0.1298  0.0212
-1.5371 -0.7257 -0.4871 -0.2359 -1.1724
[torch.FloatTensor of size 4x5]

>>> torch.mode(a, 1)
(
-2.4884
-0.7646
-2.0068
-1.5371
[torch.FloatTensor of size 4x1]
,
 3
 4
 2
 0
[torch.LongTensor of size 4x1]
)
```

torch.**norm**()

torch.**norm**(*input*, *p=2*) → float

Returns the p-norm of the `input` Tensor.

- **input** (Tensor) – the input *Tensor*
- **p** (*float, optional*) – the exponent value in the norm formulation

Example:

```
>>> a = torch.randn(1, 3)
>>> a

-0.4376 -0.5328  0.9547
[torch.FloatTensor of size 1x3]

>>> torch.norm(a, 3)
1.0338925067372466
```

`torch.`**norm**(*input*, *p*, *dim*, *out=None*) → Tensor

Returns the p-norm of each row of the `input` Tensor in the given dimension `dim`.

The output Tensor is of the same size as `input` except in the dimension `dim` where it is of size 1.

- **input** (Tensor) – the input *Tensor*
- **p** (*float*) – the exponent value in the norm formulation
- **dim** (*int*) – the dimension to reduce
- **out** (Tensor, *optional*) – the result Tensor

Example:

```
>>> a = torch.randn(4, 2)
>>> a

-0.6891 -0.6662
 0.2697  0.7412
 0.5254 -0.7402
 0.5528 -0.2399
[torch.FloatTensor of size 4x2]

>>> torch.norm(a, 2, 1)

 0.9585
 0.7888
 0.9077
 0.6026
[torch.FloatTensor of size 4x1]

>>> torch.norm(a, 0, 1)

 2
 2
 2
 2
[torch.FloatTensor of size 4x1]
```

torch.**prod**()

torch.**prod**(*input*) → float

Returns the product of all elements in the `input` Tensor.

input (Tensor) – the input *Tensor*

Example:

```
>>> a = torch.randn(1, 3)
>>> a

 0.6170  0.3546  0.0253
[torch.FloatTensor of size 1x3]

>>> torch.prod(a)
0.005537458061418483
```

torch.**prod**(*input*, *dim*, *out=None*) → Tensor

Returns the product of each row of the `input` Tensor in the given dimension `dim`.

The output Tensor is of the same size as `input` except in the dimension `dim` where it is of size 1.

- **input** (Tensor) – the input *Tensor*
- **dim** (*int*) – the dimension to reduce
- **out** (Tensor, *optional*) – the result Tensor

Example:

```
>>> a = torch.randn(4, 2)
>>> a

 0.1598 -0.6884
-0.1831 -0.4412
-0.9925 -0.6244
-0.2416 -0.8080
[torch.FloatTensor of size 4x2]

>>> torch.prod(a, 1)

-0.1100
 0.0808
 0.6197
 0.1952
[torch.FloatTensor of size 4x1]
```

torch.**std**()

torch.**std**(*input*) → float

Returns the standard-deviation of all elements in the `input` Tensor.

input (Tensor) – the input *Tensor*

Example:

```
>>> a = torch.randn(1, 3)
>>> a

-1.3063  1.4182 -0.3061
[torch.FloatTensor of size 1x3]

>>> torch.std(a)
1.3782334731508061
```

`torch.`**`std`**(*input*, *dim*, *out=None*) → Tensor

Returns the standard-deviation of each row of the `input` Tensor in the given dimension `dim`.

The output Tensor is of the same size as `input` except in the dimension `dim` where it is of size 1.

- **`input`** (`Tensor`) – the input *Tensor*
- **`dim`** (*`int`*) – the dimension to reduce
- **`out`** (`Tensor`, *`optional`*) – the result Tensor

Example:

```
>>> a = torch.randn(4, 4)
>>> a

 0.1889 -2.4856  0.0043  1.8169
-0.7701 -0.4682 -2.2410  0.4098
 0.1919 -1.1856 -1.0361  0.9085
 0.0173  1.0662  0.2143 -0.5576
[torch.FloatTensor of size 4x4]

>>> torch.std(a, dim=1)

 1.7756
 1.1025
 1.0045
 0.6725
[torch.FloatTensor of size 4x1]
```

`torch.`**`sum`**()

`torch.`**`sum`**(*input*) → float

Returns the sum of all elements in the `input` Tensor.

`input` (`Tensor`) – the input *Tensor*

Example:

```
>>> a = torch.randn(1, 3)
>>> a

 0.6170  0.3546  0.0253
[torch.FloatTensor of size 1x3]

>>> torch.sum(a)
0.9969287421554327
```

torch.**sum**(*input*, *dim*, *out=None*) → Tensor

Returns the sum of each row of the input Tensor in the given dimension dim.

The output Tensor is of the same size as input except in the dimension dim where it is of size 1.

- **input** (Tensor) – the input *Tensor*
- **dim** (*int*) – the dimension to reduce
- **out** (Tensor, *optional*) – the result Tensor

Example:

```
>>> a = torch.randn(4, 4)
>>> a

-0.4640  0.0609  0.1122  0.4784
-1.3063  1.6443  0.4714 -0.7396
-1.3561 -0.1959  1.0609 -1.9855
 2.6833  0.5746 -0.5709 -0.4430
[torch.FloatTensor of size 4x4]

>>> torch.sum(a, 1)

 0.1874
 0.0698
-2.4767
 2.2440
[torch.FloatTensor of size 4x1]
```

torch.**var**()

torch.**var**(*input*) → float

Returns the variance of all elements in the input Tensor.

input (Tensor) – the input *Tensor*

Example:

```
>>> a = torch.randn(1, 3)
>>> a

-1.3063  1.4182 -0.3061
[torch.FloatTensor of size 1x3]

>>> torch.var(a)
1.899527506513334
```

torch.**var**(*input*, *dim*, *out=None*) → Tensor

Returns the variance of each row of the input Tensor in the given dimension dim.

The output Tensor is of the same size as input except in the dimension dim where it is of size 1.

- **input** (Tensor) – the input *Tensor*
- **dim** (*int*) – the dimension to reduce

- **out** (Tensor, *optional*) – the result Tensor

Example:

```
>>> a = torch.randn(4, 4)
>>> a

-1.2738 -0.3058  0.1230 -1.9615
 0.8771 -0.5430 -0.9233  0.9879
 1.4107  0.0317 -0.6823  0.2255
-1.3854  0.4953 -0.2160  0.2435
[torch.FloatTensor of size 4x4]

>>> torch.var(a, 1)

 0.8859
 0.9509
 0.7548
 0.6949
[torch.FloatTensor of size 4x1]
```

Comparison Ops

torch.**eq**(*input*, *other*, *out=None*) → Tensor

Computes element-wise equality

The second argument can be a number or a tensor of the same shape and type as the first argument.

- **input** (Tensor) – Tensor to compare
- **other** (Tensor *or* *float*) – Tensor or value to compare
- **out** (Tensor, *optional*) – Output tensor. Must be a *ByteTensor* or the same type as *tensor*.

a `torch.ByteTensor` containing a 1 at each location where the tensors are equal and a 0 at every other location

Tensor

Example:

```
>>> torch.eq(torch.Tensor([[1, 2], [3, 4]]), torch.Tensor([[1, 1], [4, 4]]))
1  0
0  1
[torch.ByteTensor of size 2x2]
```

torch.**equal**(*tensor1*, *tensor2*) → bool

True if two tensors have the same size and elements, False otherwise.

Example:

```
>>> torch.equal(torch.Tensor([1, 2]), torch.Tensor([1, 2]))
True
```

torch.**ge**(*input*, *other*, *out=None*) → Tensor
Computes *tensor* >= *other* element-wise.

The second argument can be a number or a tensor of the same shape and type as the first argument.

- **input** (Tensor) – Tensor to compare
- **other** (Tensor *or* *float*) – Tensor or value to compare
- **out** (Tensor, *optional*) – Output tensor. Must be a *ByteTensor* or the same type as *tensor*.

a torch.ByteTensor containing a 1 at each location where comparison is true.

Tensor

Example:

```
>>> torch.ge(torch.Tensor([[1, 2], [3, 4]]), torch.Tensor([[1, 1], [4, 4]]))
 1  1
 0  1
[torch.ByteTensor of size 2x2]
```

torch.**gt**(*input*, *other*, *out=None*) → Tensor
Computes *tensor* > *other* element-wise.

The second argument can be a number or a tensor of the same shape and type as the first argument.

- **input** (Tensor) – Tensor to compare
- **other** (Tensor *or* *float*) – Tensor or value to compare
- **out** (Tensor, *optional*) – Output tensor. Must be a *ByteTensor* or the same type as *tensor*.

a torch.ByteTensor containing a 1 at each location where comparison is true.

Tensor

Example:

```
>>> torch.gt(torch.Tensor([[1, 2], [3, 4]]), torch.Tensor([[1, 1], [4, 4]]))
 0  1
 0  0
[torch.ByteTensor of size 2x2]
```

torch.**kthvalue**(*input*, *k*, *dim=None*, *out=None*) -> (*Tensor*, *LongTensor*)
Returns the k`th smallest element of the given :attr:`input Tensor along a given dimension.

If dim is not given, the last dimension of the *input* is chosen.

A tuple of *(values, indices)* is returned, where the *indices* is the indices of the kth-smallest element in the original *input* Tensor in dimention *dim*.

- **input** (Tensor) – the input *Tensor*
- **k** (*int*) – k for the k-th smallest element
- **dim** (*int*, *optional*) – The dimension to sort along

- **out** (*tuple, optional*) – The output tuple of (Tensor, LongTensor) can be optionally given to be used as output buffers

Example:

```
>>> x = torch.range(1, 5)
>>> x

 1
 2
 3
 4
 5
[torch.FloatTensor of size 5]

>>> torch.kthvalue(x, 4)
(
 4
[torch.FloatTensor of size 1]
,
 3
[torch.LongTensor of size 1]
)
```

`torch.le`(*input*, *other*, *out=None*) → Tensor

Computes *tensor* <= *other* element-wise.

The second argument can be a number or a tensor of the same shape and type as the first argument.

- **input** (Tensor) – Tensor to compare
- **other** (Tensor *or float*) – Tensor or value to compare
- **out** (Tensor, *optional*) – Output tensor. Must be a *ByteTensor* or the same type as *tensor*.

a `torch.ByteTensor` containing a 1 at each location where comparison is true.

Tensor

Example:

```
>>> torch.le(torch.Tensor([[1, 2], [3, 4]]), torch.Tensor([[1, 1], [4, 4]]))
 1  0
 1  1
[torch.ByteTensor of size 2x2]
```

`torch.lt`(*input*, *other*, *out=None*) → Tensor

Computes *tensor* < *other* element-wise.

The second argument can be a number or a tensor of the same shape and type as the first argument.

- **input** (Tensor) – Tensor to compare
- **other** (Tensor *or float*) – Tensor or value to compare
- **out** (Tensor, *optional*) – Output tensor. Must be a *ByteTensor* or the same type as *tensor*.

a `torch.ByteTensor` containing a 1 at each location where comparison is true.

Tensor

Example:

```
>>> torch.lt(torch.Tensor([[1, 2], [3, 4]]), torch.Tensor([[1, 1], [4, 4]]))
 0  0
 1  0
[torch.ByteTensor of size 2x2]
```

`torch.max()`

`torch.max(input)` → float

Returns the maximum value of all elements in the `input` Tensor.

input (`Tensor`) – the input *Tensor*

Example:

```
>>> a = torch.randn(1, 3)
>>> a

 0.4729 -0.2266 -0.2085
[torch.FloatTensor of size 1x3]

>>> torch.max(a)
0.4729
```

`torch.max(input, dim, max=None, max_indices=None) -> (Tensor, LongTensor)`

Returns the maximum value of each row of the `input` Tensor in the given dimension `dim`. Also returns the index location of each maximum value found.

The output Tensors are of the same size as `input` except in the dimension `dim` where they are of size 1.

- **input** (`Tensor`) – the input *Tensor*
- **dim** (*int*) – the dimension to reduce
- **max** (`Tensor, optional`) – the result Tensor with maximum values in dimension `dim`
- **max_indices** (`LongTensor, optional`) – the result Tensor with the index locations of the maximum values in dimension `dim`

Example:

```
>> a = torch.randn(4, 4)
>> a

0.0692  0.3142  1.2513 -0.5428
0.9288  0.8552 -0.2073  0.6409
1.0695 -0.0101 -2.4507 -1.2230
0.7426 -0.7666  0.4862 -0.6628
torch.FloatTensor of size 4x4]

>>> torch.max(a, 1)
(
 1.2513
 0.9288
 1.0695
```

```
 0.7426
[torch.FloatTensor of size 4x1]
,
 2
 0
 0
 0
[torch.LongTensor of size 4x1]
)
```

torch.**max**(*input*, *other*, *out=None*) → Tensor

Each element of the Tensor `input` is compared with the corresponding element of the Tensor `other` and an element-wise *max* is taken.

The shapes of `input` and `other` don't need to match. The total number of elements in each Tensor need to be the same.

: When the shapes do not match, the shape of `input` is used as the shape for the returned output Tensor

---

$out_i = max(tensor_i, other_i)$

- **input** (Tensor) – the input *Tensor*
- **other** (Tensor) – the second input *Tensor*
- **out** (Tensor, *optional*) – The result *Tensor*

Example:

```
>>> a = torch.randn(4)
>>> a

 1.3869
 0.3912
-0.8634
-0.5468
[torch.FloatTensor of size 4]

>>> b = torch.randn(4)
>>> b

 1.0067
-0.8010
 0.6258
 0.3627
[torch.FloatTensor of size 4]

>>> torch.max(a, b)

 1.3869
 0.3912
 0.6258
 0.3627
[torch.FloatTensor of size 4]
```

torch.**min**()

torch.**min**(*input*) → float

Returns the minimum value of all elements in the `input` Tensor.

input (Tensor) – the input *Tensor*

Example:

```
>>> a = torch.randn(1, 3)
>>> a

 0.4729 -0.2266 -0.2085
[torch.FloatTensor of size 1x3]

>>> torch.min(a)
-0.22663167119026184
```

torch.**min**(*input*, *dim*, *min=None*, *min_indices=None*) -> (*Tensor*, *LongTensor*)

Returns the minimum value of each row of the `input` Tensor in the given dimension `dim`. Also returns the index location of each minimum value found.

The output Tensors are of the same size as `input` except in the dimension `dim` where they are of size 1.

- **input** (Tensor) – the input *Tensor*
- **dim** (*int*) – the dimension to reduce
- **min** (Tensor, *optional*) – the result Tensor with minimum values in dimension `dim`
- **min_indices** (*LongTensor*, *optional*) – the result Tensor with the index locations of the minimum values in dimension `dim`

Example:

```
>> a = torch.randn(4, 4)
>> a

0.0692  0.3142  1.2513 -0.5428
0.9288  0.8552 -0.2073  0.6409
1.0695 -0.0101 -2.4507 -1.2230
0.7426 -0.7666  0.4862 -0.6628
torch.FloatTensor of size 4x4]

>> torch.min(a, 1)

0.5428
0.2073
2.4507
0.7666
torch.FloatTensor of size 4x1]

3
2
2
1
torch.LongTensor of size 4x1]
```

`torch.min(input, other, out=None) → Tensor`

Each element of the Tensor `input` is compared with the corresponding element of the Tensor `other` and an element-wise *min* is taken. The resulting Tensor is returned.

The shapes of `input` and `other` don't need to match. The total number of elements in each Tensor need to be the same.

: When the shapes do not match, the shape of `input` is used as the shape for the returned output Tensor

---

$out_i = min(tensor_i, other_i)$

- **input** (Tensor) – the input *Tensor*
- **other** (Tensor) – the second input *Tensor*
- **out** (Tensor, *optional*) – The result *Tensor*

Example:

```
>>> a = torch.randn(4)
>>> a

 1.3869
 0.3912
-0.8634
-0.5468
[torch.FloatTensor of size 4]

>>> b = torch.randn(4)
>>> b

 1.0067
-0.8010
 0.6258
 0.3627
[torch.FloatTensor of size 4]

>>> torch.min(a, b)

 1.0067
-0.8010
-0.8634
-0.5468
[torch.FloatTensor of size 4]
```

`torch.ne(input, other, out=None) → Tensor`

Computes *tensor != other* element-wise.

The second argument can be a number or a tensor of the same shape and type as the first argument.

- **input** (Tensor) – Tensor to compare
- **other** (Tensor *or float*) – Tensor or value to compare
- **out** (Tensor, *optional*) – Output tensor. Must be a *ByteTensor* or the same type as *tensor*.

a `torch.ByteTensor` containing a 1 at each location where comparison is true.

Tensor

Example:

```
>>> torch.ne(torch.Tensor([[1, 2], [3, 4]]), torch.Tensor([[1, 1], [4, 4]]))
 0  1
 1  0
[torch.ByteTensor of size 2x2]
```

`torch.`**`sort`**(*input*, *dim=None*, *descending=False*, *out=None*) -> (*Tensor*, *LongTensor*)

Sorts the elements of the `input` Tensor along a given dimension in ascending order by value.

If `dim` is not given, the last dimension of the *input* is chosen.

If `descending` is *True* then the elements are sorted in descending order by value.

A tuple of (sorted_tensor, sorted_indices) is returned, where the sorted_indices are the indices of the elements in the original *input* Tensor.

- **input** (`Tensor`) – the input *Tensor*
- **dim** (*int, optional*) – The dimension to sort along
- **descending** (*bool, optional*) – Controls the sorting order (ascending or descending)
- **out** (*tuple, optional*) – The output tuple of (Tensor, LongTensor) can be optionally given to be used as output buffers

Example:

```
>>> x = torch.randn(3, 4)
>>> sorted, indices = torch.sort(x)
>>> sorted

-1.6747  0.0610  0.1190  1.4137
-1.4782  0.7159  1.0341  1.3678
-0.3324 -0.0782  0.3518  0.4763
[torch.FloatTensor of size 3x4]

>>> indices

 0  1  3  2
 2  1  0  3
 3  1  0  2
[torch.LongTensor of size 3x4]

>>> sorted, indices = torch.sort(x, 0)
>>> sorted

-1.6747 -0.0782 -1.4782 -0.3324
 0.3518  0.0610  0.4763  0.1190
 1.0341  0.7159  1.4137  1.3678
[torch.FloatTensor of size 3x4]

>>> indices

 0  2  1  2
```

```
 2  0  2  0
 1  1  0  1
[torch.LongTensor of size 3x4]
```

torch.**topk**(*input*, *k*, *dim=None*, *largest=True*, *sorted=True*, *out=None*) -> (*Tensor*, *LongTensor*)

Returns the `k` largest elements of the given `input` Tensor along a given dimension.

If `dim` is not given, the last dimension of the *input* is chosen.

If `largest` is *False* then the *k* smallest elements are returned.

A tuple of *(values, indices)* is returned, where the *indices* are the indices of the elements in the original *input* Tensor.

The boolean option `sorted` if *True*, will make sure that the returned *k* elements are themselves sorted

- **input** (`Tensor`) – the input *Tensor*
- **k** (*int*) – the k in "top-k"
- **dim** (*int, optional*) – The dimension to sort along
- **largest** (*bool, optional*) – Controls whether to return largest or smallest elements
- **sorted** (*bool, optional*) – Controls whether to return the elements in sorted order
- **out** (*tuple, optional*) – The output tuple of (Tensor, LongTensor) can be optionally given to be used as output buffers

Example:

```
>>> x = torch.range(1, 5)
>>> x

 1
 2
 3
 4
 5
[torch.FloatTensor of size 5]

>>> torch.topk(x, 3)
(
 5
 4
 3
[torch.FloatTensor of size 3]
,
 4
 3
 2
[torch.LongTensor of size 3]
)
>>> torch.topk(x, 3, 0, largest=False)
(
 1
 2
 3
[torch.FloatTensor of size 3]
,
```

```
 0
 1
 2
[torch.LongTensor of size 3]
)
```

Other Operations

`torch.`**`cross`**(*input*, *other*, *dim=-1*, *out=None*) → Tensor

Returns the cross product of vectors in dimension `dim` of `input` and `other`.

`input` and `other` must have the same size, and the size of their `dim` dimension should be 3.

If `dim` is not given, it defaults to the first dimension found with the size 3.

- **`input`** (`Tensor`) – the input *Tensor*
- **`other`** (`Tensor`) – the second input *Tensor*
- **`dim`** (*`int, optional`*) – the dimension to take the cross-product in.
- **`out`** (`Tensor, `*`optional`*) – The result *Tensor*

Example:

```
>>> a = torch.randn(4, 3)
>>> a

-0.6652 -1.0116 -0.6857
 0.2286  0.4446 -0.5272
 0.0476  0.2321  1.9991
 0.6199  1.1924 -0.9397
[torch.FloatTensor of size 4x3]

>>> b = torch.randn(4, 3)
>>> b

-0.1042 -1.1156  0.1947
 0.9947  0.1149  0.4701
-1.0108  0.8319 -0.0750
 0.9045 -1.3754  1.0976
[torch.FloatTensor of size 4x3]

>>> torch.cross(a, b, dim=1)

-0.9619  0.2009  0.6367
 0.2696 -0.6318 -0.4160
-1.6805 -2.0171  0.2741
 0.0163 -1.5304 -1.9311
[torch.FloatTensor of size 4x3]

>>> torch.cross(a, b)

-0.9619  0.2009  0.6367
 0.2696 -0.6318 -0.4160
-1.6805 -2.0171  0.2741
```

```
 0.0163 -1.5304 -1.9311
[torch.FloatTensor of size 4x3]
```

torch.**diag**(*input*, *diagonal=0*, *out=None*) → Tensor

- If `input` is a vector (1D Tensor), then returns a 2D square Tensor with the elements of `input` as the diagonal.
- If `input` is a matrix (2D Tensor), then returns a 1D Tensor with the diagonal elements of `input`.

The argument `diagonal` controls which diagonal to consider.

- `diagonal` = 0, is the main diagonal.
- `diagonal` > 0, is above the main diagonal.
- `diagonal` < 0, is below the main diagonal.

- **input** (Tensor) – the input *Tensor*
- **diagonal** (*int, optional*) – the diagonal to consider
- **out** (Tensor, *optional*) – The result *Tensor*

Example:

Get the square matrix where the input vector is the diagonal:

```
>>> a = torch.randn(3)
>>> a

 1.0480
-2.3405
-1.1138
[torch.FloatTensor of size 3]

>>> torch.diag(a)

 1.0480  0.0000  0.0000
 0.0000 -2.3405  0.0000
 0.0000  0.0000 -1.1138
[torch.FloatTensor of size 3x3]

>>> torch.diag(a, 1)

 0.0000  1.0480  0.0000  0.0000
 0.0000  0.0000 -2.3405  0.0000
 0.0000  0.0000  0.0000 -1.1138
 0.0000  0.0000  0.0000  0.0000
[torch.FloatTensor of size 4x4]
```

Get the k-th diagonal of a given matrix:

```
>>> a = torch.randn(3, 3)
>>> a

-1.5328 -1.3210 -1.5204
 0.8596  0.0471 -0.2239
-0.6617  0.0146 -1.0817
```

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

>>> torch.diag(a, 0)

-1.5328
 0.0471
-1.0817
[torch.FloatTensor of size 3]

>>> torch.diag(a, 1)

-1.3210
-0.2239
[torch.FloatTensor of size 2]
```

torch.**histc**(*input*, *bins=100*, *min=0*, *max=0*, *out=None*) → Tensor

Computes the histogram of a tensor.

The elements are sorted into equal width bins between *min* and *max*. If *min* and *max* are both zero, the minimum and maximum values of the data are used.

- **input** (Tensor) – Input data
- **bins** (*int*) – Number of histogram bins
- **min** (*int*) – Lower end of the range (inclusive)
- **max** (*int*) – Upper end of the range (inclusive)
- **out** (Tensor, *optional*) – Output argument

the histogram

Tensor

Example:

```
>>> torch.histc(torch.FloatTensor([1, 2, 1]), bins=4, min=0, max=3)
FloatTensor([0, 2, 1, 0])
```

torch.**renorm**(*input*, *p*, *dim*, *maxnorm*, *out=None*) → Tensor

Returns a Tensor where each sub-tensor of `input` along dimension `dim` is normalized such that the *p*-norm of the sub-tensor is lower than the value `maxnorm`

: If the norm of a row is lower than *maxnorm*, the row is unchanged

- **input** (Tensor) – The input Tensor
- **p** (*float*) – The power for the norm computation
- **dim** (*int*) – The dimension to slice over to get the sub-tensors
- **maxnorm** (*float*) – The maximum norm to keep each sub-tensor under
- **out** (Tensor, *optional*) – Output tensor

Example:

```
>>> x = torch.ones(3, 3)
>>> x[1].fill_(2)
>>> x[2].fill_(3)
>>> x

 1  1  1
 2  2  2
 3  3  3
[torch.FloatTensor of size 3x3]

>>> torch.renorm(x, 1, 0, 5)

 1.0000  1.0000  1.0000
 1.6667  1.6667  1.6667
 1.6667  1.6667  1.6667
[torch.FloatTensor of size 3x3]
```

`torch.`**`trace`**(*input*) → float

Returns the sum of the elements of the diagonal of the input 2D matrix.

Example:

```
>>> x = torch.range(1, 9).view(3, 3)
>>> x

 1  2  3
 4  5  6
 7  8  9
[torch.FloatTensor of size 3x3]

>>> torch.trace(x)
15.0
```

`torch.`**`tril`**(*input*, *k=0*, *out=None*) → Tensor

Returns the lower triangular part of the matrix (2D Tensor) `input`, the other elements of the result Tensor `out` are set to 0.

The lower triangular part of the matrix is defined as the elements on and below the diagonal.

The argument `k` controls which diagonal to consider.

- `k` = 0, is the main diagonal.
- `k` > 0, is above the main diagonal.
- `k` < 0, is below the main diagonal.

- **`input`** (`Tensor`) – the input *Tensor*
- **`k`** (*`int, optional`*) – the diagonal to consider
- **`out`** (`Tensor, optional`) – The result *Tensor*

Example:

```
>>> a = torch.randn(3,3)
>>> a
```

```
 1.3225  1.7304  1.4573
-0.3052 -0.3111 -0.1809
 1.2469  0.0064 -1.6250
[torch.FloatTensor of size 3x3]

>>> torch.tril(a)

 1.3225  0.0000  0.0000
-0.3052 -0.3111  0.0000
 1.2469  0.0064 -1.6250
[torch.FloatTensor of size 3x3]

>>> torch.tril(a, k=1)

 1.3225  1.7304  0.0000
-0.3052 -0.3111 -0.1809
 1.2469  0.0064 -1.6250
[torch.FloatTensor of size 3x3]

>>> torch.tril(a, k=-1)

 0.0000  0.0000  0.0000
-0.3052  0.0000  0.0000
 1.2469  0.0064  0.0000
[torch.FloatTensor of size 3x3]
```

torch.**triu**(*input*, *k=0*, *out=None*) → Tensor

Returns the upper triangular part of the matrix (2D Tensor) `input`, the other elements of the result Tensor `out` are set to 0.

The upper triangular part of the matrix is defined as the elements on and above the diagonal.

The argument `k` controls which diagonal to consider.

- `k` = 0, is the main diagonal.
- `k` > 0, is above the main diagonal.
- `k` < 0, is below the main diagonal.

- **input** (Tensor) – the input *Tensor*
- **k** (*int, optional*) – the diagonal to consider
- **out** (Tensor, *optional*) – The result *Tensor*

Example:

```
>>> a = torch.randn(3,3)
>>> a

 1.3225  1.7304  1.4573
-0.3052 -0.3111 -0.1809
 1.2469  0.0064 -1.6250
[torch.FloatTensor of size 3x3]

>>> torch.triu(a)
```

```
 1.3225  1.7304  1.4573
 0.0000 -0.3111 -0.1809
 0.0000  0.0000 -1.6250
[torch.FloatTensor of size 3x3]

>>> torch.triu(a, k=1)

 0.0000  1.7304  1.4573
 0.0000  0.0000 -0.1809
 0.0000  0.0000  0.0000
[torch.FloatTensor of size 3x3]

>>> torch.triu(a, k=-1)

 1.3225  1.7304  1.4573
-0.3052 -0.3111 -0.1809
 0.0000  0.0064 -1.6250
[torch.FloatTensor of size 3x3]
```

BLAS and LAPACK Operations

`torch.`**`addbmm`**(*beta=1*, *mat*, *alpha=1*, *batch1*, *batch2*, *out=None*) → Tensor

Performs a batch matrix-matrix product of matrices stored in `batch1` and `batch2`, with a reduced add step (all matrix multiplications get accumulated along the first dimension). `mat` is added to the final result.

`batch1` and `batch2` must be 3D Tensors each containing the same number of matrices.

If `batch1` is a *b x n x m* Tensor, `batch2` is a *b x m x p* Tensor, `out` and `mat` will be *n x p* Tensors.

In other words, $res = (beta * M) + (alpha * sum(batch1_i @ batch2_i, i = 0, b))$

For inputs of type *FloatTensor* or *DoubleTensor*, args *beta* and *alpha* must be real numbers, otherwise they should be integers

- **beta** (*Number, optional*) – multiplier for `mat`
- **mat** (Tensor) – matrix to be added
- **alpha** (*Number, optional*) – multiplier for *batch1 @ batch2*
- **batch1** (Tensor) – First batch of matrices to be multiplied
- **batch2** (Tensor) – Second batch of matrices to be multiplied
- **out** (Tensor, *optional*) – Output tensor

Example:

```
>>> M = torch.randn(3, 5)
>>> batch1 = torch.randn(10, 3, 4)
>>> batch2 = torch.randn(10, 4, 5)
>>> torch.addbmm(M, batch1, batch2)

 -3.1162  11.0071   7.3102   0.1824  -7.6892
  1.8265   6.0739   0.4589  -0.5641  -5.4283
 -9.3387  -0.1794  -1.2318  -6.8841  -4.7239
[torch.FloatTensor of size 3x5]
```

torch.**addmm**(*beta=1*, *mat*, *alpha=1*, *mat1*, *mat2*, *out=None*) → Tensor

Performs a matrix multiplication of the matrices `mat1` and `mat2`. The matrix `mat` is added to the final result.

If `mat1` is a *n x m* Tensor, `mat2` is a *m x p* Tensor, `out` and `mat` will be *n x p* Tensors.

alpha and *beta* are scaling factors on *mat1 @ mat2* and *mat* respectively.

In other words, $out = (beta * M) + (alpha * mat1@mat2)$

For inputs of type *FloatTensor* or *DoubleTensor*, args `beta` and `alpha` must be real numbers, otherwise they should be integers

- **beta** (*Number, optional*) – multiplier for `mat`
- **mat** (Tensor) – matrix to be added
- **alpha** (*Number, optional*) – multiplier for *mat1 @ mat2*
- **mat1** (Tensor) – First matrix to be multiplied
- **mat2** (Tensor) – Second matrix to be multiplied
- **out** (Tensor, *optional*) – Output tensor

Example:

```
>>> M = torch.randn(2, 3)
>>> mat1 = torch.randn(2, 3)
>>> mat2 = torch.randn(3, 3)
>>> torch.addmm(M, mat1, mat2)

-0.4095 -1.9703  1.3561
 5.7674 -4.9760  2.7378
[torch.FloatTensor of size 2x3]
```

torch.**addmv**(*beta=1*, *tensor*, *alpha=1*, *mat*, *vec*, *out=None*) → Tensor

Performs a matrix-vector product of the matrix `mat` and the vector `vec`. The vector `tensor` is added to the final result.

If `mat` is a *n x m* Tensor, `vec` is a 1D Tensor of size *m*, `out` and `tensor` will be 1D of size *n*.

alpha and *beta* are scaling factors on *mat * vec* and *tensor* respectively.

In other words:

$out = (beta * tensor) + (alpha * (mat@vec2))$

For inputs of type *FloatTensor* or *DoubleTensor*, args `beta` and `alpha` must be real numbers, otherwise they should be integers

- **beta** (*Number, optional*) – multiplier for `tensor`
- **tensor** (Tensor) – vector to be added
- **alpha** (*Number, optional*) – multiplier for *mat @ vec*
- **mat** (Tensor) – matrix to be multiplied
- **vec** (Tensor) – vector to be multiplied
- **out** (Tensor, *optional*) – Output tensor

Example:

```
>>> M = torch.randn(2)
>>> mat = torch.randn(2, 3)
>>> vec = torch.randn(3)
>>> torch.addmv(M, mat, vec)

-2.0939
-2.2950
[torch.FloatTensor of size 2]
```

`torch.`**`addr`**(*beta=1*, *mat*, *alpha=1*, *vec1*, *vec2*, *out=None*) → Tensor

Performs the outer-product of vectors `vec1` and `vec2` and adds it to the matrix `mat`.

Optional values `beta` and `alpha` are scalars that multiply `mat` and $(vec1 \otimes vec2)$ respectively

In other words, $out = (beta * mat) + (alpha * vec1 \otimes vec2)$

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 x m*

For inputs of type *FloatTensor* or *DoubleTensor*, args `beta` and `alpha` must be real numbers, otherwise they should be integers

- **beta** (*Number, optional*) – Multiplier for `mat`
- **mat** (`Tensor`) – Matrix to be added
- **alpha** (*Number, optional*) – Multiplier for outer product of for `vec1` and `vec2`
- **vec1** (`Tensor`) – First vector of the outer product
- **vec2** (`Tensor`) – Second vector of the outer product
- **out** (`Tensor, optional`) – Output tensor

Example:

```
>>> vec1 = torch.range(1, 3)
>>> vec2 = torch.range(1, 2)
>>> M = torch.zeros(3, 2)
>>> torch.addr(M, vec1, vec2)
 1  2
 2  4
 3  6
[torch.FloatTensor of size 3x2]
```

`torch.`**`baddbmm`**(*beta=1*, *mat*, *alpha=1*, *batch1*, *batch2*, *out=None*) → Tensor

Performs a batch matrix-matrix product of matrices in `batch1` and `batch2`. `mat` is added to the final result.

`batch1` and `batch2` must be 3D Tensors each containing the same number of matrices.

If `batch1` is a *b x n x m* Tensor, `batch2` is a *b x m x p* Tensor, `out` and `mat` will be *b x n x p* Tensors.

In other words, $res_i = (beta * M_i) + (alpha * batch1_i \times batch2_i)$

For inputs of type *FloatTensor* or *DoubleTensor*, args `beta` and `alpha` must be real numbers, otherwise they should be integers

- **beta** (*Number, optional*) – multiplier for `mat`
- **mat** (`Tensor`) – tensor to be added
- **alpha** (*Number, optional*) – multiplier for *batch1 @ batch2*

- **batch1** (Tensor) – First batch of matrices to be multiplied
- **batch2** (Tensor) – Second batch of matrices to be multiplied
- **out** (Tensor, *optional*) – Output tensor

Example:

```
>>> M = torch.randn(10, 3, 5)
>>> batch1 = torch.randn(10, 3, 4)
>>> batch2 = torch.randn(10, 4, 5)
>>> torch.baddbmm(M, batch1, batch2).size()
torch.Size([10, 3, 5])
```

torch.**bmm**(*batch1*, *batch2*, *out=None*) → Tensor

Performs a batch matrix-matrix product of matrices stored in batch1 and batch2.

batch1 and batch2 must be 3D Tensors each containing the same number of matrices.

If batch1 is a *b x n x m* Tensor, batch2 is a *b x m x p* Tensor, out will be a *b x n x p* Tensor.

- **batch1** (Tensor) – First batch of matrices to be multiplied
- **batch2** (Tensor) – Second batch of matrices to be multiplied
- **out** (Tensor, *optional*) – Output tensor

Example:

```
>>> batch1 = torch.randn(10, 3, 4)
>>> batch2 = torch.randn(10, 4, 5)
>>> res = torch.bmm(batch1, batch2)
>>> res.size()
torch.Size([10, 3, 5])
```

torch.**btrifact**(*A*, *info=None*) → Tensor, IntTensor

Batch LU factorization.

Returns a tuple containing the LU factorization and pivots. The optional argument *info* provides information if the factorization succeeded for each minibatch example. The info values are from dgetrf and a non-zero value indicates an error occurred. The specific values are from cublas if cuda is being used, otherwise LAPACK.

A (Tensor) – tensor to factor.

Example:

```
>>> A = torch.randn(2, 3, 3)
>>> A_LU = A.btrifact()
```

torch.**btrisolve**(*b*, *LU_data*, *LU_pivots*) → Tensor

Batch LU solve.

Returns the LU solve of the linear system Ax = b.

- **b** (Tensor) – RHS tensor.
- **LU_data** (Tensor) – Pivoted LU factorization of A from btrifact.
- **LU_pivots** (*IntTensor*) – Pivots of the LU factorization.

Example:

```
>>> A = torch.randn(2, 3, 3)
>>> b = torch.randn(2, 3)
>>> A_LU_data, A_LU_pivots, info = torch.btrifact(A)
>>> x = b.trisolve(A_LU_data, A_LU_pivots)
>>> torch.norm(A.bmm(x.unsqueeze(2)) - b)
6.664001874625056e-08
```

`torch.`**`dot`**(*tensor1*, *tensor2*) → float

Computes the dot product (inner product) of two tensors. Both tensors are treated as 1-D vectors.

Example:

```
>>> torch.dot(torch.Tensor([2, 3]), torch.Tensor([2, 1]))
7.0
```

`torch.`**`eig`**(*a*, *eigenvectors=False*, *out=None)* -> *(Tensor*, *Tensor*)

Computes the eigenvalues and eigenvectors of a real square matrix.

- **a** (`Tensor`) – A square matrix for which the eigenvalues and eigenvectors will be computed
- **eigenvectors** (*bool*) – *True* to compute both eigenvalues and eigenvectors. Otherwise, only eigenvalues will be computed.
- **out** (*tuple, optional*) – Output tensors

tuple containing

- **e** (*Tensor*): the right eigenvalues of `a`
- **v** (*Tensor*): the eigenvectors of `a` if `eigenvectors` is ``True``; otherwise an empty tensor

(*Tensor*, *Tensor*)

`torch.`**`gels`**(*B*, *A*, *out=None*) → Tensor

Computes the solution to the least squares and least norm problems for a full rank m by n matrix A.

If $m >= n$, *gels()* solves the least-squares problem:

$$\begin{array}{ll} \text{minimize} & \|AX - B\|_F. \end{array}$$

If $m < n$, *gels()* solves the least-norm problem:

$$\begin{array}{ll} \text{minimize} & \|X\|_F \\ \text{subject to} & AX = B. \end{array}$$

The first n rows of the returned matrix X contains the solution. The remaining rows contain residual information: the euclidean norm of each column starting at row n is the residual for the corresponding column.

- **B** (`Tensor`) – The matrix B
- **A** (`Tensor`) – The m by n matrix A
- **out** (*tuple, optional*) – Optional destination tensor

tuple containing:

- **X** (*Tensor*): the least squares solution
- **qr** (*Tensor*): the details of the QR factorization

(*Tensor*, *Tensor*)

: The returned matrices will always be tranposed, irrespective of the strides of the input matrices. That is, they will have stride *(1, m)* instead of *(m, 1)*.

Example:

```
>>> A = torch.Tensor([[1, 1, 1],
...                   [2, 3, 4],
...                   [3, 5, 2],
...                   [4, 2, 5],
...                   [5, 4, 3]])
>>> B = torch.Tensor([[-10, -3],
                      [ 12, 14],
                      [ 14, 12],
                      [ 16, 16],
                      [ 18, 16]])
>>> X, _ = torch.gels(B, A)
>>> X
2.0000  1.0000
1.0000  1.0000
1.0000  2.0000
[torch.FloatTensor of size 3x2]
```

`torch.`**`geqrf`**(*input*, *out=None*) -> (*Tensor*, *Tensor*)

This is a low-level function for calling LAPACK directly.

You'll generally want to use *`torch.qr()`* instead.

Computes a QR decomposition of `input`, 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.

- **`input`** (`Tensor`) – the input matrix
- **`out`** (*`tuple, optional`*) – The result tuple of (Tensor, Tensor)

`torch.`**`ger`**(*vec1*, *vec2*, *out=None*) → Tensor

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

- **`vec1`** (`Tensor`) – 1D input vector
- **`vec2`** (`Tensor`) – 1D input vector
- **`out`** (*`Tensor, optional`*) – optional output matrix

Example:

```
>>> v1 = torch.range(1, 4)
>>> v2 = torch.range(1, 3)
>>> torch.ger(v1, v2)

  1   2   3
  2   4   6
  3   6   9
  4   8  12
[torch.FloatTensor of size 4x3]
```

torch.**gesv**(*B*, *A*, *out=None*) -> (*Tensor*, *Tensor*)

X, LU = torch.gesv(B, A) returns the solution to the system of linear equations represented by $AX = B$

LU contains *L* and *U* factors for LU factorization of *A*.

A has to be a square and non-singular matrix (2D Tensor).

If *A* is an *m x m* matrix and *B* is *m x k*, the result *LU* is *m x m* and *X* is *m x k* .

: Irrespective of the original strides, the returned matrices *X* and *LU* will be transposed, i.e. with strides *(1, m)* instead of *(m, 1)*.

- **B** (Tensor) – input matrix of *m x k* dimensions
- **A** (Tensor) – input square matrix of *m x m* dimensions
- **out** (Tensor, *optional*) – optional output matrix

Example:

```
>>> 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()
>>> X, LU = torch.gesv(B, A)
>>> torch.dist(B, torch.mm(A, X))
9.250057093890353e-06
```

torch.**inverse**(*input*, *out=None*) → Tensor

Takes the inverse of the square matrix `input`.

: Irrespective of the original strides, the returned matrix will be transposed, i.e. with strides *(1, m)* instead of *(m, 1)*

- **input** (Tensor) – the input 2D square *Tensor*
- **out** (Tensor, *optional*) – the optional output *Tensor*

Example:

```
>>> x = torch.rand(10, 10)
>>> x

 0.7800  0.2267  0.7855  0.9479  0.5914  0.7119  0.4437  0.9131  0.1289  0.1982
 0.0045  0.0425  0.2229  0.4626  0.6210  0.0207  0.6338  0.7067  0.6381  0.8196
 0.8350  0.7810  0.8526  0.9364  0.7504  0.2737  0.0694  0.5899  0.8516  0.3883
 0.6280  0.6016  0.5357  0.2936  0.7827  0.2772  0.0744  0.2627  0.6326  0.9153
 0.7897  0.0226  0.3102  0.0198  0.9415  0.9896  0.3528  0.9397  0.2074  0.6980
 0.5235  0.6119  0.6522  0.3399  0.3205  0.5555  0.8454  0.3792  0.4927  0.6086
 0.1048  0.0328  0.5734  0.6318  0.9802  0.4458  0.0979  0.3320  0.3701  0.0909
 0.2616  0.3485  0.4370  0.5620  0.5291  0.8295  0.7693  0.1807  0.0650  0.8497
 0.1655  0.2192  0.6913  0.0093  0.0178  0.3064  0.6715  0.5101  0.2561  0.3396
 0.4370  0.4695  0.8333  0.1180  0.4266  0.4161  0.0699  0.4263  0.8865  0.2578
[torch.FloatTensor of size 10x10]

>>> x = torch.rand(10, 10)
>>> y = torch.inverse(x)
>>> z = torch.mm(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.FloatTensor of size 10x10]

>>> torch.max(torch.abs(z - torch.eye(10))) # Max nonzero
5.096662789583206e-07
```

`torch.`**`mm`**(*mat1*, *mat2*, *out=None*) → Tensor

Performs a matrix multiplication of the matrices `mat1` and `mat2`.

If `mat1` is a *n x m* Tensor, `mat2` is a *m x p* Tensor, `out` will be a *n x p* Tensor.

- **mat1** (Tensor) – First matrix to be multiplied
- **mat2** (Tensor) – Second matrix to be multiplied
- **out** (Tensor, *optional*) – Output tensor

Example:

```
>>> mat1 = torch.randn(2, 3)
>>> mat2 = torch.randn(3, 3)
>>> torch.mm(mat1, mat2)
 0.0519 -0.3304  1.2232
 4.3910 -5.1498  2.7571
[torch.FloatTensor of size 2x3]
```

`torch.`**`mv`**(*mat*, *vec*, *out=None*) → Tensor

Performs a matrix-vector product of the matrix `mat` and the vector `vec`.

If `mat` is a *n x m* Tensor, `vec` is a 1D Tensor of size *m*, `out` will be 1D of size *n*.

- **mat** (Tensor) – matrix to be multiplied
- **vec** (Tensor) – vector to be multiplied
- **out** (Tensor, *optional*) – Output tensor

Example:

```
>>> mat = torch.randn(2, 3)
>>> vec = torch.randn(3)
>>> torch.mv(mat, vec)
-2.0939
-2.2950
[torch.FloatTensor of size 2]
```

torch.**orgqr**()

torch.**ormqr**()

torch.**potrf**()

torch.**potri**()

torch.**potrs**()

torch.**pstrf**()

torch.**qr**(*input*, *out=None*) -> (*Tensor*, *Tensor*)

Computes the QR decomposition of a matrix `input`: returns matrices *q* and *r* such that $x = q * r$, with *q* being an orthogonal matrix and *r* being an upper triangular matrix.

This returns the thin (reduced) QR factorization.

: precision may be lost if the magnitudes of the elements of *input* are large

---

: 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.

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

- **input** (Tensor) – the input 2D *Tensor*
- **out** (*tuple*, *optional*) – A tuple of Q and R Tensors

Example:

```
>>> a = torch.Tensor([[12, -51, 4], [6, 167, -68], [-4, 24, -41]])
>>> 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.FloatTensor of size 3x3]

>>> r

 -14.0000  -21.0000   14.0000
   0.0000 -175.0000   70.0000
   0.0000    0.0000  -35.0000
[torch.FloatTensor of size 3x3]

>>> torch.mm(q, r).round()

  12  -51    4
   6  167  -68
  -4   24  -41
[torch.FloatTensor of size 3x3]

>>> torch.mm(q.t(), q).round()

 1 -0  0
-0  1  0
 0  0  1
[torch.FloatTensor of size 3x3]
```

torch.**svd**(*input*, *some=True*, *out=None)* -> *(Tensor*, *Tensor*, *Tensor*)

U, S, V = torch.svd(A) returns the singular value decomposition of a real matrix *A* of size *(n x m)* such that $A = USV'*$.

U is of shape *n x n*

S is of shape *n x m*

V is of shape *m x m*.

`some` represents the number of singular values to be computed. If *some=True*, it computes some and *some=False* computes all.

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

- **input** (*Tensor*) – the input 2D Tensor
- **some** (*bool, optional*) – controls the number of singular values to be computed
- **out** (*tuple, optional*) – the result tuple

Example:

```
>>> 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
[torch.FloatTensor of size 6x5]

>>> 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.FloatTensor of size 6x5]

>>> s

 27.4687
 22.6432
  8.5584
  5.9857
  2.0149
[torch.FloatTensor of size 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 -0.4932 -0.6227 -0.4396
[torch.FloatTensor of size 5x5]

>>> torch.dist(a, torch.mm(torch.mm(u, torch.diag(s)), v.t()))
8.934150226306685e-06
```

torch.**symeig**(*input*, *eigenvectors=False*, *upper=True*, *out=None*) -> *(Tensor*, *Tensor)*

e, V = torch.symeig(input) returns eigenvalues and eigenvectors of a symmetric real matrix `input`.

input and *V* are *m x m* matrices and *e* is a *m* dimensional vector.

This function calculates all eigenvalues (and vectors) of *input* such that *input = V diag(e) V'*

The boolean argument `eigenvectors` defines computation of eigenvectors or eigenvalues only.

If it is *False*, only eigenvalues are computed. If it is *True*, both eigenvalues and eigenvectors are computed.

Since the input matrix *input* is supposed to be symmetric, only the upper triangular portion is used by default.

If `upper` is *False*, 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)*.

- **input** (Tensor) – the input symmetric matrix
- **eigenvectors** (*boolean, optional*) – controls whether eigenvectors have to be computed
- **upper** (*boolean, optional*) – controls whether to consider upper-triangular or lower-triangular region
- **out** (*tuple, optional*) – The result tuple of (Tensor, Tensor)

Examples:

```
>>> 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()

>>> e, v = torch.symeig(a, eigenvectors=True)
>>> e

-11.0656
 -6.2287
  0.8640
  8.8655
 16.0948
[torch.FloatTensor of size 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.FloatTensor of size 5x5]
```

torch.**trtrs**()

CHAPTER 7

torch.Tensor

A *torch.Tensor* is a multi-dimensional matrix containing elements of a single data type.

Torch defines seven CPU tensor types and eight GPU tensor types:

Data type	CPU tensor	GPU tensor
32-bit floating point	`torch.FloatTensor`	`torch.cuda.FloatTensor`
64-bit floating point	`torch.DoubleTensor`	`torch.cuda.DoubleTensor`
16-bit floating point	N/A	`torch.cuda.HalfTensor`
8-bit integer (unsigned)	`torch.ByteTensor`	`torch.cuda.ByteTensor`
8-bit integer (signed)	`torch.CharTensor`	`torch.cuda.CharTensor`
16-bit integer (signed)	`torch.ShortTensor`	`torch.cuda.ShortTensor`
32-bit integer (signed)	`torch.IntTensor`	`torch.cuda.IntTensor`
64-bit integer (signed)	`torch.LongTensor`	`torch.cuda.LongTensor`

The *torch.Tensor* constructor is an alias for the default tensor type (`torch.FloatTensor`).

A tensor can be constructed from a Python `list` or sequence:

```
>>> torch.FloatTensor([[1, 2, 3], [4, 5, 6]])
1  2  3
4  5  6
[torch.FloatTensor of size 2x3]
```

An empty tensor can be constructed by specifying its size:

```
>>> torch.IntTensor(2, 4).zero_()
0  0  0  0
0  0  0  0
[torch.IntTensor of size 2x4]
```

The contents of a tensor can be accessed and modified using Python's indexing and slicing notation:

```
>>> x = torch.FloatTensor([[1, 2, 3], [4, 5, 6]])
>>> print(x[1][2])
6.0
```

```
>>> x[0][1] = 8
>>> print(x)
 1  8  3
 4  5  6
[torch.FloatTensor of size 2x3]
```

Each tensor has an associated `torch.Storage`, which holds its data. The tensor class provides multi-dimensional, strided view of a storage and defines numeric operations on it.

: Methods which mutate a tensor are marked with an underscore suffix. For example, `torch.FloatTensor.abs_()` computes the absolute value in-place and returns the modified tensor, while `torch.FloatTensor.abs()` computes the result in a new tensor.

---

class `torch.`**Tensor**
class `torch.`**Tensor**(**sizes*)
class `torch.`**Tensor**(*size*)
class `torch.`**Tensor**(*sequence*)
class `torch.`**Tensor**(*ndarray*)
class `torch.`**Tensor**(*tensor*)
class `torch.`**Tensor**(*storage*)

Creates a new tensor from an optional size or data.

If no arguments are given, an empty zero-dimensional tensor is returned. If a `numpy.ndarray`, *`torch.Tensor`*, or `torch.Storage` is given, a new tensor that shares the same data is returned. If a Python sequence is given, a new tensor is created from a copy of the sequence.

abs() → Tensor
See *`torch.abs()`*

abs_() → Tensor
In-place version of *`abs()`*

acos() → Tensor
See *`torch.acos()`*

acos_() → Tensor
In-place version of *`acos()`*

add(*value*)
See *`torch.add()`*

add_(*value*)
In-place version of *`add()`*

addbmm(*beta=1*, *mat*, *alpha=1*, *batch1*, *batch2*) → Tensor
See *`torch.addbmm()`*

addbmm_(*beta=1*, *mat*, *alpha=1*, *batch1*, *batch2*) → Tensor
In-place version of *`addbmm()`*

addcdiv(*value=1*, *tensor1*, *tensor2*) → Tensor
See *`torch.addcdiv()`*

addcdiv_(*value=1*, *tensor1*, *tensor2*) → Tensor
In-place version of *`addcdiv()`*

addcmul(*value=1*, *tensor1*, *tensor2*) → Tensor
See *`torch.addcmul()`*

addcmul_(*value=1*, *tensor1*, *tensor2*) → Tensor
In-place version of *addcmul()*

addmm(*beta=1*, *mat*, *alpha=1*, *mat1*, *mat2*) → Tensor
See *torch.addmm()*

addmm_(*beta=1*, *mat*, *alpha=1*, *mat1*, *mat2*) → Tensor
In-place version of *addmm()*

addmv(*beta=1*, *tensor*, *alpha=1*, *mat*, *vec*) → Tensor
See *torch.addmv()*

addmv_(*beta=1*, *tensor*, *alpha=1*, *mat*, *vec*) → Tensor
In-place version of *addmv()*

addr(*beta=1*, *alpha=1*, *vec1*, *vec2*) → Tensor
See *torch.addr()*

addr_(*beta=1*, *alpha=1*, *vec1*, *vec2*) → Tensor
In-place version of *addr()*

apply_(*callable*) → Tensor
Applies the function `callable` to each element in the tensor, replacing each element with the value returned by `callable`.

: This function only works with CPU tensors and should not be used in code sections that require high performance.

---

asin() → Tensor
See *torch.asin()*

asin_() → Tensor
In-place version of *asin()*

atan() → Tensor
See *torch.atan()*

atan2(*other*) → Tensor
See *torch.atan2()*

atan2_(*other*) → Tensor
In-place version of *atan2()*

atan_() → Tensor
In-place version of *atan()*

baddbmm(*beta=1*, *alpha=1*, *batch1*, *batch2*) → Tensor
See *torch.baddbmm()*

baddbmm_(*beta=1*, *alpha=1*, *batch1*, *batch2*) → Tensor
In-place version of *baddbmm()*

bernoulli() → Tensor
See *torch.bernoulli()*

bernoulli_() → Tensor
In-place version of *bernoulli()*

bmm(*batch2*) → Tensor
See *torch.bmm()*

byte()
Casts this tensor to byte type

cauchy_(*median=0*, *sigma=1*, *, *generator=None*) → Tensor
Fills the tensor with numbers drawn from the Cauchy distribution:

$$P(x) = \frac{1}{\pi} \frac{\sigma}{(x - median)^2 + \sigma^2}$$

ceil() → Tensor
See *torch.ceil()*

ceil_() → Tensor
In-place version of *ceil()*

char()
Casts this tensor to char type

chunk(*n_chunks*, *dim=0*)
Splits this tensor into a tuple of tensors.

See *torch.chunk()*.

clamp(*min*, *max*) → Tensor
See *torch.clamp()*

clamp_(*min*, *max*) → Tensor
In-place version of *clamp()*

clone() → Tensor
Returns a copy of the tensor. The copy has the same size and data type as the original tensor.

contiguous() → Tensor
Returns a contiguous Tensor containing the same data as this tensor. If this tensor is contiguous, this function returns the original tensor.

copy_(*src*, *async=False*) → Tensor
Copies the elements from `src` into this tensor and returns this tensor.

The source tensor should have the same number of elements as this tensor. It may be of a different data type or reside on a different device.

- **src** (Tensor) – Source tensor to copy
- **async** (*bool*) – If True and this copy is between CPU and GPU, then the copy may occur asynchronously with respect to the host. For other copies, this argument has no effect.

cos() → Tensor
See *torch.cos()*

cos_() → Tensor
In-place version of *cos()*

cosh() → Tensor
See *torch.cosh()*

cosh_() → Tensor
In-place version of *cosh()*

cpu()
Returns a CPU copy of this tensor if it's not already on the CPU

cross(*other*, *dim=-1*) → Tensor
See *torch.cross()*

cuda(*device=None*, *async=False*)
Returns a copy of this object in CUDA memory.

If this object is already in CUDA memory and on the correct device, then no copy is performed and the original object is returned.

- **device** (*int*) – The destination GPU id. Defaults to the current device.
- **async** (*bool*) – If True and the source is in pinned memory, the copy will be asynchronous with respect to the host. Otherwise, the argument has no effect.

cumprod(*dim*) → Tensor
See *torch.cumprod()*

cumsum(*dim*) → Tensor
See *torch.cumsum()*

data_ptr() → int
Returns the address of the first element of this tensor.

diag(*diagonal=0*) → Tensor
See *torch.diag()*

dim() → int
Returns the number of dimensions of this tensor.

dist(*other*, *p=2*) → Tensor
See *torch.dist()*

div(*value*)
See *torch.div()*

div_(*value*)
In-place version of *div()*

dot(*tensor2*) → float
See *torch.dot()*

double()
Casts this tensor to double type

eig(*eigenvectors=False*) -> (*Tensor*, *Tensor*)
See *torch.eig()*

element_size() → int
Returns the size in bytes of an individual element.

Example

```
>>> torch.FloatTensor().element_size()
4
>>> torch.ByteTensor().element_size()
1
```

eq(*other*) → Tensor
See *torch.eq()*

eq_(*other*) → Tensor
In-place version of *eq()*

equal(*other*) → bool
See *torch.equal()*

exp() → Tensor
See *torch.exp()*

exp_() → Tensor
In-place version of *exp()*

expand(**sizes*)
Returns a new view of the tensor with singleton dimensions expanded to a larger size.

Tensor can be also expanded to a larger number of dimensions, and the new ones will be appended at the front.

Expanding a tensor does not allocate new memory, but only creates a new view on the existing tensor where a dimension of size one is expanded to a larger size by setting the `stride` to 0. Any dimension of size 1 can be expanded to an arbitrary value without allocating new memory.

***sizes** (*torch.Size or int...*) – The desired expanded size

Example

```
>>> x = torch.Tensor([[1], [2], [3]])
>>> x.size()
torch.Size([3, 1])
>>> x.expand(3, 4)
 1  1  1  1
 2  2  2  2
 3  3  3  3
[torch.FloatTensor of size 3x4]
```

expand_as(*tensor*)
Expands this tensor to the size of the specified tensor.

This is equivalent to:

```
self.expand(tensor.size())
```

exponential_(*lambd=1*, *, *generator=None*) → Tensor
Fills this tensor with elements drawn from the exponential distribution:

$$P(x) = \lambda e^{-\lambda x}$$

fill_(*value*) → Tensor
Fills this tensor with the specified value.

float()
Casts this tensor to float type

floor() → Tensor
See *torch.floor()*

floor_() → Tensor
In-place version of *floor()*

fmod(*divisor*) → Tensor
See *torch.fmod()*

fmod_(*divisor*) → Tensor
In-place version of *fmod()*

frac() → Tensor
See *torch.frac()*

frac_() → Tensor
In-place version of *frac()*

gather(*dim*, *index*) → Tensor
See *torch.gather()*

ge(*other*) → Tensor
See *torch.ge()*

ge_(*other*) → Tensor
In-place version of *ge()*

gels(*A*) → Tensor
See *torch.gels()*

geometric_(*p*, *, *generator=None*) → Tensor
Fills this tensor with elements drawn from the geometric distribution:

$$P(X = k) = (1 - p)^{k-1} p$$

geqrf() -> (*Tensor*, *Tensor*)
See *torch.geqrf()*

ger(*vec2*) → Tensor
See *torch.ger()*

gesv(*A*) → Tensor, Tensor
See *torch.gesv()*

gt(*other*) → Tensor
See *torch.gt()*

gt_(*other*) → Tensor
In-place version of *gt()*

half()
Casts this tensor to half-precision float type

histc(*bins=100*, *min=0*, *max=0*) → Tensor
See *torch.histc()*

index(*m*) → Tensor
Selects elements from this tensor using a binary mask or along a given dimension. The expression `tensor.index(m)` is equivalent to `tensor[m]`.

m (*int or ByteTensor or slice*) – The dimension or mask used to select elements

index_add_(*dim*, *index*, *tensor*) → 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 raised.

- **dim** (*int*) – Dimension along which to index

- **index** (*LongTensor*) – Indices to select from tensor
- **tensor** (Tensor) – Tensor containing values to add

Example

```
>>> x = torch.Tensor([[1, 1, 1], [1, 1, 1], [1, 1, 1]])
>>> t = torch.Tensor([[1, 2, 3], [4, 5, 6], [7, 8, 9]])
>>> index = torch.LongTensor([0, 2, 1])
>>> x.index_add_(0, index, t)
>>> x
  2   3   4
  8   9  10
  5   6   7
[torch.FloatTensor of size 3x3]
```

index_copy_(*dim*, *index*, *tensor*) → 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 raised.

- **dim** (*int*) – Dimension along which to index
- **index** (*LongTensor*) – Indices to select from tensor
- **tensor** (Tensor) – Tensor containing values to copy

Example

```
>>> x = torch.Tensor(3, 3)
>>> t = torch.Tensor([[1, 2, 3], [4, 5, 6], [7, 8, 9]])
>>> index = torch.LongTensor([0, 2, 1])
>>> x.index_copy_(0, index, t)
>>> x
 1  2  3
 7  8  9
 4  5  6
[torch.FloatTensor of size 3x3]
```

index_fill_(*dim*, *index*, *val*) → Tensor

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

- **dim** (*int*) – Dimension along which to index
- **index** (*LongTensor*) – Indices
- **val** (*float*) – Value to fill

Example

```
>>> x = torch.Tensor([[1, 2, 3], [4, 5, 6], [7, 8, 9]])
>>> index = torch.LongTensor([0, 2])
>>> x.index_fill_(1, index, -1)
```

```
>>> x
-1  2 -1
-1  5 -1
-1  8 -1
[torch.FloatTensor of size 3x3]
```

index_select(*dim*, *index*) → Tensor
See *torch.index_select()*

int()
Casts this tensor to int type

inverse() → Tensor
See *torch.inverse()*

is_contiguous() → bool
Returns True if this tensor is contiguous in memory in C order.

is_cuda

is_pinned()
Returns true if this tensor resides in pinned memory

is_set_to(*tensor*) → bool
Returns True if this object refers to the same `THTensor` object from the Torch C API as the given tensor.

is_signed()

kthvalue(*k*, *dim=None*) -> (*Tensor*, *LongTensor*)
See *torch.kthvalue()*

le(*other*) → Tensor
See *torch.le()*

le_(*other*) → Tensor
In-place version of *le()*

lerp(*start*, *end*, *weight*)
See *torch.lerp()*

lerp_(*start*, *end*, *weight*)
In-place version of *lerp()*

log() → Tensor
See *torch.log()*

log1p() → Tensor
See *torch.log1p()*

log1p_() → Tensor
In-place version of *log1p()*

log_() → Tensor
In-place version of *log()*

log_normal_(*mean=1*, *std=2*, ***, *generator=None*)
Fills this tensor with numbers samples from the log-normal distribution parameterized by the given mean (μ) and standard deviation (σ). Note that *mean* and `stdv` are the mean and standard deviation of the underlying normal distribution, and not of the returned distribution:

$$P(x) = \frac{1}{x\sigma\sqrt{2\pi}} e^{-\frac{(\ln x - \mu)^2}{2\sigma^2}}$$

long()
Casts this tensor to long type

lt(*other*) → Tensor
See *torch.lt()*

lt_(*other*) → Tensor
In-place version of *lt()*

map_(*tensor*, *callable*)
Applies `callable` for each element in this tensor and the given tensor and stores the results in this tensor. The `callable` should have the signature:

```
def callable(a, b) -> number
```

masked_copy_(*mask*, *source*)
Copies elements from `source` into this tensor at positions where the `mask` is one. The `mask` should have the same number of elements as this tensor. The `source` should have at least as many elements as the number of ones in `mask`

- **mask** (*ByteTensor*) – The binary mask
- **source** (Tensor) – The tensor to copy from

: The `mask` operates on the `self` tensor, not on the given `source` tensor.

masked_fill_(*mask*, *value*)
Fills elements of this tensor with `value` where `mask` is one. The `mask` should have the same number of elements as this tensor, but the shape may differ.

- **mask** (*ByteTensor*) – The binary mask
- **value** (Tensor) – The value to fill

masked_select(*mask*) → Tensor
See *torch.masked_select()*

max(*dim=None) -> float or (Tensor*, *Tensor*)
See *torch.max()*

mean(*dim=None) -> float or (Tensor*, *Tensor*)
See *torch.mean()*

median(*dim=-1*, *values=None*, *indices=None) -> (Tensor*, *LongTensor*)
See *torch.median()*

min(*dim=None) -> float or (Tensor*, *Tensor*)
See *torch.min()*

mm(*mat2*) → Tensor
See *torch.mm()*

mode(*dim=-1*, *values=None*, *indices=None) -> (Tensor*, *LongTensor*)
See *torch.mode()*

mul(*value*) → Tensor
See *torch.mul()*

mul_(*value*)
In-place version of *mul()*

multinomial(*num_samples*, *replacement=False*, *, *generator=None*)
See *torch.multinomial()*

mv(*vec*) → Tensor
See *torch.mv()*

narrow(*dimension*, *start*, *length*) → Tensor
Returns a new tensor that is a narrowed version of this tensor. The dimension *dim* is narrowed from `start` to `start + length`. The returned tensor and this tensor share the same underlying storage.

- **dimension** (*int*) – The dimension along which to narrow
- **start** (*int*) – The starting dimension
- **length** (*int*) –

Example

```
>>> x = torch.Tensor([[1, 2, 3], [4, 5, 6], [7, 8, 9]])
>>> x.narrow(0, 0, 2)
 1  2  3
 4  5  6
[torch.FloatTensor of size 2x3]
>>> x.narrow(1, 1, 2)
 2  3
 5  6
 8  9
[torch.FloatTensor of size 3x2]
```

ndimension() → int
Alias for *dim()*

ne(*other*) → Tensor
See *torch.ne()*

ne_(*other*) → Tensor
In-place version of *ne()*

neg() → Tensor
See *torch.neg()*

neg_() → Tensor
In-place version of *neg()*

nelement() → int
Alias for *numel()*

new(**args*, ***kwargs*)
Constructs a new tensor of the same data type.

nonzero() → LongTensor
See *torch.nonzero()*

norm(*p=2*) → float
See *torch.norm()*

normal_(*mean=0*, *std=1*, *, *generator=None*)
Fills this tensor with elements samples from the normal distribution parameterized by *mean* and *std*.

numel() → int
See *torch.numel()*

numpy() → ndarray
Returns this tensor as a NumPy `ndarray`. This tensor and the returned `ndarray` share the same underlying storage. Changes to this tensor will be reflected in the `ndarray` and vice versa.

orgqr(*input2*) → Tensor
See *torch.orgqr()*

ormqr(*input2*, *input3*, *left=True*, *transpose=False*) → Tensor
See *torch.ormqr()*

permute(**dims*)
Permute the dimensions of this tensor.

***dims** (*int...*) – The desired ordering of dimensions

Example

```
>>> x = torch.randn(2, 3, 5)
>>> x.size()
torch.Size([2, 3, 5])
>>> x.permute(2, 0, 1).size()
torch.Size([5, 2, 3])
```

pin_memory()
Copies the tensor to pinned memory, if it's not already pinned.

potrf(*upper=True*) → Tensor
See *torch.potrf()*

potri(*upper=True*) → Tensor
See *torch.potri()*

potrs(*input2*, *upper=True*) → Tensor
See *torch.potrs()*

pow(*exponent*)
See *torch.pow()*

pow_(*exponent*)
In-place version of *pow()*

prod() → float
See *torch.prod()*

pstrf(*upper=True*, *tol=-1*) -> (*Tensor*, *IntTensor*)
See *torch.pstrf()*

qr() -> (*Tensor*, *Tensor*)
See *torch.qr()*

random_(*from=0*, *to=None*, *, *generator=None*)
Fills this tensor with numbers sampled from the uniform distribution or discrete uniform distribution over [from, to - 1]. If not specified, the values are only bounded by this tensor's data type.

reciprocal() → Tensor
See *torch.reciprocal()*

reciprocal_() → Tensor
In-place version of *reciprocal()*

remainder(*divisor*) → Tensor
See *torch.remainder()*

remainder_(*divisor*) → Tensor
In-place version of *remainder()*

renorm(*p*, *dim*, *maxnorm*) → Tensor
See *torch.renorm()*

renorm_(*p*, *dim*, *maxnorm*) → Tensor
In-place version of *renorm()*

repeat(**sizes*)
Repeats this tensor along the specified dimensions.

Unlike *expand()*, this function copies the tensor's data.

> ***sizes** (*torch.Size or int...*) – The number of times to repeat this tensor along each dimension

Example

```
>>> x = torch.Tensor([1, 2, 3])
>>> x.repeat(4, 2)
 1  2  3  1  2  3
 1  2  3  1  2  3
 1  2  3  1  2  3
 1  2  3  1  2  3
[torch.FloatTensor of size 4x6]
>>> x.repeat(4, 2, 1).size()
torch.Size([4, 2, 3])
```

resize_(**sizes*)
Resizes this tensor to the specified size. If the number of elements is larger than the current storage size, then the underlying storage is resized to fit the new number of elements. If the number of elements is smaller, the underlying storage is not changed. Existing elements are preserved but any new memory is uninitialized.

> **sizes** (*torch.Size or int...*) – The desired size

Example

```
>>> x = torch.Tensor([[1, 2], [3, 4], [5, 6]])
>>> x.resize_(2, 2)
>>> x
 1  2
 3  4
[torch.FloatTensor of size 2x2]
```

resize_as_(*tensor*)
Resizes the current tensor to be the same size as the specified tensor. This is equivalent to:

```
self.resize_(tensor.size())
```

round() → Tensor
See *torch.round()*

round_() → Tensor
In-place version of *round()*

rsqrt() → Tensor
See *torch.rsqrt()*

rsqrt_() → Tensor
In-place version of *rsqrt()*

scatter_(*input*, *dim*, *index*, *src*) → Tensor
Writes all values from the Tensor `src` into self at the indices specified in the *index* Tensor. 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 *0* and *(self.size(dim) -1)* inclusive and all values in a row along the specified dimension must be unique.

- **input** (*Tensor*) – The source tensor
- **dim** (*int*) – The axis along which to index
- **index** (*LongTensor*) – The indices of elements to scatter
- **src** (*Tensor or float*) – The source element(s) to scatter

Example:

```
>>> x = torch.rand(2, 5)
>>> x

 0.4319  0.6500  0.4080  0.8760  0.2355
 0.2609  0.4711  0.8486  0.8573  0.1029
[torch.FloatTensor of size 2x5]

>>> torch.zeros(3, 5).scatter_(0, torch.LongTensor([[0, 1, 2, 0, 0], [2, 0, 0,
↪ 1, 2]]), x)

 0.4319  0.4711  0.8486  0.8760  0.2355
 0.0000  0.6500  0.0000  0.8573  0.0000
 0.2609  0.0000  0.4080  0.0000  0.1029
[torch.FloatTensor of size 3x5]

>>> z = torch.zeros(2, 4).scatter_(1, torch.LongTensor([[2], [3]]), 1.23)
>>> z

 0.0000  0.0000  1.2300  0.0000
 0.0000  0.0000  0.0000  1.2300
[torch.FloatTensor of size 2x4]
```

select(*dim*, *index*) → Tensor or number
Slices the tensor along the selected dimension at the given index. If this tensor is one dimensional, this function returns a number. Otherwise, it returns a tensor with the given dimension removed.

- **dim** (*int*) – Dimension to slice

- **index** (*int*) – Index to select

: *select()* is equivalent to slicing. For example, `tensor.select(0, index)` is equivalent to `tensor[index]` and `tensor.select(2, index)` is equivalent to `tensor[:,:,index]`.

---

set_(*source=None*, *storage_offset=0*, *size=None*, *stride=None*)
Sets the underlying storage, size, and strides. If `source` is a tensor, this tensor will share the same storage and have the same size and strides as the given tensor. Changes to elements in one tensor will be reflected in the other.

If `source` is a `Storage`, the method sets the underlying storage, offset, size, and stride.

- **source** (*Tensor or Storage*) – The tensor or storage to use
- **storage_offset** (*int*) – The offset in the storage
- **size** (*torch.Size*) – The desired size. Defaults to the size of the source.
- **stride** (*tuple*) – The desired stride. Defaults to C-contiguous strides.

share_memory_()
Moves the underlying storage to shared memory.

This is a no-op if the underlying storage is already in shared memory and for CUDA tensors. Tensors in shared memory cannot be resized.

short()
Casts this tensor to short type

sigmoid() → Tensor
See *torch.sigmoid()*

sigmoid_() → Tensor
In-place version of *sigmoid()*

sign() → Tensor
See *torch.sign()*

sign_() → Tensor
In-place version of *sign()*

sin() → Tensor
See *torch.sin()*

sin_() → Tensor
In-place version of *sin()*

sinh() → Tensor
See *torch.sinh()*

sinh_() → Tensor
In-place version of *sinh()*

size() → torch.Size
Returns the size of the tensor. The returned value is a subclass of `tuple`.

Example

```
>>> torch.Tensor(3, 4, 5).size()
torch.Size([3, 4, 5])
```

sort(*dim=None*, *descending=False*) -> (*Tensor*, *LongTensor*)
See *torch.sort()*

split(*split_size*, *dim=0*)
Splits this tensor into a tuple of tensors.

See *torch.split()*.

sqrt() → Tensor
See *torch.sqrt()*

sqrt_() → Tensor
In-place version of *sqrt()*

squeeze(*dim=None*)
See *torch.squeeze()*

squeeze_(*dim=None*)
In-place version of *squeeze()*

std() → float
See *torch.std()*

storage() → torch.Storage
Returns the underlying storage

storage_offset() → int
Returns this tensor's offset in the underlying storage in terms of number of storage elements (not bytes).

Example

```
>>> x = torch.Tensor([1, 2, 3, 4, 5])
>>> x.storage_offset()
0
>>> x[3:].storage_offset()
3
```

classmethod storage_type()

stride() → tuple
Returns the stride of the tensor.

sub(*value*, *other*) → Tensor
Subtracts a scalar or tensor from this tensor. If both `value` and `other` are specified, each element of `other` is scaled by `value` before being used.

sub_(*x*) → Tensor
In-place version of *sub()*

sum(*dim=None*) → float
See *torch.sum()*

svd(*some=True*) -> (*Tensor*, *Tensor*, *Tensor*)
See *torch.svd()*

symeig(*eigenvectors=False*, *upper=True) -> (Tensor*, *Tensor*)
See *torch.symeig()*

t() → Tensor
See *torch.t()*

t_() → Tensor
In-place version of *t()*

tan() → Tensor
See *torch.tan()*

tan_() → Tensor
In-place version of *tan()*

tanh() → Tensor
See *torch.tanh()*

tanh_() → Tensor
In-place version of *tanh()*

tolist()
Returns a nested list represenation of this tensor.

topk(*k*, *dim=None*, *largest=True*, *sorted=True) -> (Tensor*, *LongTensor*)
See *torch.topk()*

trace() → float
See *torch.trace()*

transpose(*dim0*, *dim1*) → Tensor
See *torch.transpose()*

transpose_(*dim0*, *dim1*) → Tensor
In-place version of *transpose()*

tril(*k=0*) → Tensor
See *torch.tril()*

tril_(*k=0*) → Tensor
In-place version of *tril()*

triu(*k=0*) → Tensor
See *torch.triu()*

triu_(*k=0*) → Tensor
In-place version of *triu()*

trtrs(*A*, *upper=True*, *transpose=False*, *unitriangular=False) -> (Tensor*, *Tensor*)
See *torch.trtrs()*

trunc() → Tensor
See *torch.trunc()*

trunc_() → Tensor
In-place version of *trunc()*

type(*new_type=None*, *async=False*)
Casts this object to the specified type.

If this is already of the correct type, no copy is performed and the original object is returned.

- **new_type** (*type or string*) – The desired type

- **async** (*bool*) – If True, and the source is in pinned memory and destination is on the GPU or vice versa, the copy is performed asynchronously with respect to the host. Otherwise, the argument has no effect.

type_as(*tensor*)

Returns this tensor cast to the type of the given tensor.

This is a no-op if the tensor is already of the correct type. This is equivalent to:

```
self.type(tensor.type())
```

Params: tensor (Tensor): the tensor which has the desired type

unfold(*dim*, *size*, *step*) → Tensor

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.

- **dim** (*int*) – dimension in which unfolding happens
- **size** (*int*) – size of each slice that is unfolded
- **step** (*int*) – the step between each slice

Example:

```
>>> x = torch.range(1, 7)
>>> x

 1
 2
 3
 4
 5
 6
 7
[torch.FloatTensor of size 7]

>>> x.unfold(0, 2, 1)

 1  2
 2  3
 3  4
 4  5
 5  6
 6  7
[torch.FloatTensor of size 6x2]

>>> x.unfold(0, 2, 2)

 1  2
 3  4
 5  6
[torch.FloatTensor of size 3x2]
```

uniform_ (*from=0*, *to=1*) → Tensor
Fills this tensor with numbers sampled from the uniform distribution:

unsqueeze (*dim*)
See *torch.unsqueeze()*

unsqueeze_ (*dim*)
In-place version of *unsqueeze()*

var () → float
See *torch.var()*

view (**args*) → Tensor
Returns a new tensor with the same data but different size.

The returned tensor shares the same data and must have the same number of elements, but may have a different size. A tensor must be *contiguous()* to be viewed.

args (*torch.Size or int...*) – Desired size

Example

```
>>> x = torch.randn(4, 4)
>>> x.size()
torch.Size([4, 4])
>>> y = x.view(16)
>>> y.size()
torch.Size([16])
>>> z = x.view(-1, 8)  # the size -1 is inferred from other dimensions
>>> z.size()
torch.Size([2, 8])
```

view_as (*tensor*)
Returns this tensor viewed as the size as the specified tensor.

This is equivalent to:

```
self.view(tensor.size())
```

zero_ ()
Fills this tensor with zeros.

CHAPTER 8

torch.Storage

A `torch.Storage` is a contiguous, one-dimensional array of a single data type.

Every *torch.Tensor* has a corresponding storage of the same data type.

class `torch.`**FloatStorage**

byte()
Casts this storage to byte type

char()
Casts this storage to char type

clone()
Returns a copy of this storage

copy_()

cpu()
Returns a CPU copy of this storage if it's not already on the CPU

cuda(*device=None*, *async=False*)
Returns a copy of this object in CUDA memory.

If this object is already in CUDA memory and on the correct device, then no copy is performed and the original object is returned.

- **device** (*int*) – The destination GPU id. Defaults to the current device.
- **async** (*bool*) – If True and the source is in pinned memory, the copy will be asynchronous with respect to the host. Otherwise, the argument has no effect.

data_ptr()

double()
Casts this storage to double type

element_size()

fill_()

float()
Casts this storage to float type

from_buffer()

half()
Casts this storage to half type

int()
Casts this storage to int type

is_cuda = False

is_pinned()

is_shared()

is_sparse = False

long()
Casts this storage to long type

new()

pin_memory()
Copies the storage to pinned memory, if it's not already pinned.

resize_()

share_memory_()
Moves the storage to shared memory.

This is a no-op for storages already in shared memory and for CUDA storages, which do not need to be moved for sharing across processes. Storages in shared memory cannot be resized.

Returns: self

short()
Casts this storage to short type

size()

tolist()
Returns a list containing the elements of this storage

type(*new_type=None*, *async=False*)
Casts this object to the specified type.

If this is already of the correct type, no copy is performed and the original object is returned.

- **new_type** (*type or string*) – The desired type
- **async** (*bool*) – If True, and the source is in pinned memory and destination is on the GPU or vice versa, the copy is performed asynchronously with respect to the host. Otherwise, the argument has no effect.

CHAPTER 9

torch.nn

Parameters

class torch.nn.**Parameter**

A kind of Variable that is to be considered a module parameter.

Parameters are *Variable* subclasses, that have a very special property when used with *Module* s - when they're assigned as Module attributes they are automatically added to the list of its parameters, and will appear e.g. in *parameters()* iterator. Assigning a Variable doesn't have such effect. This is because one might want to cache some temporary state, like last hidden state of the RNN, in the model. If there was no such class as *Parameter*, these temporaries would get registered too.

Another difference is that parameters can't be volatile and that they require gradient by default.

- **data** (Tensor) – parameter tensor.
- **requires_grad** (*bool, optional*) – if the parameter requires gradient. See *Excluding subgraphs from backward* for more details.

Containers

Module

class torch.nn.**Module**

Base class for all neural network modules.

Your models should also subclass this class.

Modules can also contain other Modules, allowing to nest them in a tree structure. You can assign the submodules as regular attributes:

```
import torch.nn as nn
import torch.nn.functional as F

class Model(nn.Module):
    def __init__(self):
        super(Model, self).__init__()
        self.conv1 = nn.Conv2d(1, 20, 5)
        self.conv2 = nn.Conv2d(20, 20, 5)

    def forward(self, x):
       x = F.relu(self.conv1(x))
       return F.relu(self.conv2(x))
```

Submodules assigned in this way will be registered, and will have their parameters converted too when you call .cuda(), etc.

add_module(*name*, *module*)
Adds a child module to the current module.

The module can be accessed as an attribute using the given name.

children()
Returns an iterator over immediate children modules.

cpu(*device_id=None*)
Moves all model parameters and buffers to the CPU.

cuda(*device_id=None*)
Moves all model parameters and buffers to the GPU.

device_id (*int, optional*) – if specified, all parameters will be copied to that device

double()
Casts all parameters and buffers to double datatype.

eval()
Sets the module in evaluation mode.

This has any effect only on modules such as Dropout or BatchNorm.

float()
Casts all parameters and buffers to float datatype.

forward(**input*)
Defines the computation performed at every call.

Should be overriden by all subclasses.

half()
Casts all parameters and buffers to half datatype.

load_state_dict(*state_dict*)
Copies parameters and buffers from *state_dict* into this module and its descendants. The keys of *state_dict* must exactly match the keys returned by this module's *state_dict()* function.

state_dict (*dict*) – A dict containing parameters and persistent buffers.

modules()
Returns an iterator over all modules in the network.

: Duplicate modules are returned only once. In the following example, `l` will be returned only once.

```
>>> l = nn.Linear(2, 2)
>>> net = nn.Sequential(l, l)
>>> for idx, m in enumerate(net.modules()):
>>>     print(idx, '->', m)
0 -> Sequential (
  (0): Linear (2 -> 2)
  (1): Linear (2 -> 2)
)
1 -> Linear (2 -> 2)
```

---

named_children()
Returns an iterator over immediate children modules, yielding both the name of the module as well as the module itself.

Example

```
>>> for name, module in model.named_children():
>>>     if name in ['conv4', 'conv5']:
>>>         print(module)
```

named_modules(*memo=None*, *prefix=''*)
Returns an iterator over all modules in the network, yielding both the name of the module as well as the module itself.

: Duplicate modules are returned only once. In the following example, `l` will be returned only once.

```
>>> l = nn.Linear(2, 2)
>>> net = nn.Sequential(l, l)
>>> for idx, m in enumerate(net.named_modules()):
>>>     print(idx, '->', m)
0 -> ('', Sequential (
  (0): Linear (2 -> 2)
  (1): Linear (2 -> 2)
))
1 -> ('0', Linear (2 -> 2))
```

---

parameters(*memo=None*)
Returns an iterator over module parameters.

This is typically passed to an optimizer.

Example

```
>>> for param in model.parameters():
>>>     print(type(param.data), param.size())
<class 'torch.FloatTensor'> (20L,)
<class 'torch.FloatTensor'> (20L, 1L, 5L, 5L)
```

register_backward_hook(*hook*)
Registers a backward hook on the module.

The hook will be called every time the gradients with respect to module inputs are computed. The hook should have the following signature:

```
hook(module, grad_input, grad_output) -> Tensor or None
```

The `grad_input` and `grad_output` may be tuples if the module has multiple inputs or outputs. The hook should not modify its arguments, but it can optionally return a new gradient with respect to input that will be used in place of `grad_input` in subsequent computations.

This function returns a handle with a method `handle.remove()` that removes the hook from the module.

register_buffer(*name*, *tensor*)
Adds a persistent buffer to the module.

This is typically used to register a buffer that should not to be considered a model parameter. For example, BatchNorm's `running_mean` is not a parameter, but is part of the persistent state.

Buffers can be accessed as attributes using given names.

Example

```
>>> self.register_buffer('running_mean', torch.zeros(num_features))
```

register_forward_hook(*hook*)
Registers a forward hook on the module.

The hook will be called every time *forward()* computes an output. It should have the following signature:

```
hook(module, input, output) -> None
```

The hook should not modify the input or output. This function returns a handle with a method `handle.remove()` that removes the hook from the module.

register_parameter(*name*, *param*)
Adds a parameter to the module.

The parameter can be accessed as an attribute using given name.

state_dict(*destination=None*, *prefix=''*)
Returns a dictionary containing a whole state of the module.

Both parameters and persistent buffers (e.g. running averages) are included. Keys are corresponding parameter and buffer names.

Example

```
>>> module.state_dict().keys()
['bias', 'weight']
```

train(*mode=True*)
Sets the module in training mode.

This has any effect only on modules such as Dropout or BatchNorm.

zero_grad()
Sets gradients of all model parameters to zero.

Sequential

class torch.nn.**Sequential**(**args*)

A sequential container. Modules will be added to it in the order they are passed in the constructor. Alternatively, an ordered dict of modules can also be passed in.

To make it easier to understand, given is a small example:

```
# Example of using Sequential
model = nn.Sequential(
          nn.Conv2d(1,20,5),
          nn.ReLU(),
          nn.Conv2d(20,64,5),
          nn.ReLU()
        )

# Example of using Sequential with OrderedDict
model = nn.Sequential(OrderedDict([
          ('conv1', nn.Conv2d(1,20,5)),
          ('relu1', nn.ReLU()),
          ('conv2', nn.Conv2d(20,64,5)),
          ('relu2', nn.ReLU())
        ]))
```

ModuleList

class torch.nn.**ModuleList**(*modules=None*)

Holds submodules in a list.

ModuleList can be indexed like a regular Python list, but modules it contains are properly registered, and will be visible by all Module methods.

modules (*list, optional*) – a list of modules to add

Example:

```
class MyModule(nn.Module):
    def __init__(self):
        super(MyModule, self).__init__()
        self.linears = nn.ModuleList([nn.Linear(10, 10) for i in range(10)])

    def forward(self, x):
        # ModuleList can act as an iterable, or be indexed using ints
        for i, l in enumerate(self.linears):
            x = self.linears[i // 2](x) + l(x)
        return x
```

append(*module*)

Appends a given module at the end of the list.

module (nn.Module) – module to append

extend(*modules*)

Appends modules from a Python list at the end.

modules (*list*) – list of modules to append

ParameterList

class torch.nn.**ParameterList**(*parameters=None*)

Holds submodules in a list.

ParameterList can be indexed like a regular Python list, but parameters it contains are properly registered, and will be visible by all Module methods.

modules (*list, optional*) – a list of nn.Parameter` to add

Example:

```
class MyModule(nn.Module):
    def __init__(self):
        super(MyModule, self).__init__()
        self.params = nn.ParameterList([nn.Parameter(torch.randn(10, 10)) for i
↪in range(10)])

    def forward(self, x):
        # ModuleList can act as an iterable, or be indexed using ints
        for i, p in enumerate(self.params):
            x = self.params[i // 2].mm(x) + p.mm(x)
        return x
```

append(*parameter*)

Appends a given parameter at the end of the list.

parameter (nn.Parameter) – parameter to append

extend(*parameters*)

Appends parameters from a Python list at the end.

parameters (*list*) – list of parameters to append

Convolution Layers

Conv1d

class torch.nn.**Conv1d**(*in_channels, out_channels, kernel_size, stride=1, padding=0, dilation=1, groups=1, bias=True*)

Applies a 1D convolution over an input signal composed of several input planes.

In the simplest case, the output value of the layer with input size (N, C_{in}, L) and output (N, C_{out}, L_{out}) can be precisely described as:

$$out(N_i, C_{out_j}) = bias(C_{out_j}) + \sum_{k=0}^{C_{in}-1} weight(C_{out_j}, k) \star input(N_i, k)$$

where $\star$ is the valid cross-correlation operator

`stride` controls the stride for the cross-correlation.

If `padding` is non-zero, then the input is implicitly zero-padded on both sides for `padding` number of points

`dilation` controls the spacing between the kernel points. It is harder to describe, but this link has a nice visualization of what `dilation` does.

`groups` controls the connections between inputs and outputs.

At groups=1, all inputs are convolved to all outputs.

At groups=2, the operation becomes equivalent to having two conv layers side by side, each seeing half the input channels, and producing half the output channels, and both subsequently concatenated.

: Depending of the size of your kernel, several (of the last) columns of the input might be lost, because it is a valid cross-correlation, and not a full cross-correlation. It is up to the user to add proper padding.

- **in_channels** (*int*) – Number of channels in the input image
- **out_channels** (*int*) – Number of channels produced by the convolution
- **kernel_size** (*int or tuple*) – Size of the convolving kernel
- **stride** (*int or tuple, optional*) – Stride of the convolution
- **padding** (*int or tuple, optional*) – Zero-padding added to both sides of the input
- **dilation** (*int or tuple, optional*) – Spacing between kernel elements
- **groups** (*int, optional*) – Number of blocked connections from input channels to output channels
- **bias** (*bool, optional*) – If True, adds a learnable bias to the output

Shape:

- Input: (N, C_{in}, L_{in})
- Output: (N, C_{out}, L_{out}) where $L_{out} = floor((L_{in} + 2 * padding - dilation * (kernel\_size - 1) - 1)/stride + 1)$

- **weight** (Tensor) – the learnable weights of the module of shape (out_channels, in_channels, kernel_size)
- **bias** (Tensor) – the learnable bias of the module of shape (out_channels)

Examples:

```
>>> m = nn.Conv1d(16, 33, 3, stride=2)
>>> input = autograd.Variable(torch.randn(20, 16, 50))
>>> output = m(input)
```

Conv2d

class torch.nn.**Conv2d**(*in_channels*, *out_channels*, *kernel_size*, *stride=1*, *padding=0*, *dilation=1*, *groups=1*, *bias=True*)

Applies a 2D convolution over an input signal composed of several input planes.

In the simplest case, the output value of the layer with input size (N, C_{in}, H, W) and output $(N, C_{out}, H_{out}, W_{out})$ can be precisely described as:

$$out(N_i, C_{out_j}) = bias(C_{out_j}) + \sum_{k=0}^{C_{in}-1} weight(C_{out_j}, k) \star input(N_i, k)$$

where $\star$ is the valid 2D cross-correlation operator

`stride` controls the stride for the cross-correlation.

If `padding` is non-zero, then the input is implicitly zero-padded on both sides for `padding` number of points

`dilation` controls the spacing between the kernel points. It is harder to describe, but this link has a nice visualization of what `dilation` does.

`groups` controls the connections between inputs and outputs.

At groups=1, all inputs are convolved to all outputs.

At groups=2, the operation becomes equivalent to having two conv layers side by side, each seeing half the input channels, and producing half the output channels, and both subsequently concatenated.

The parameters `kernel_size`, `stride`, `padding`, `dilation` can either be:

- a single `int` – in which case the same value is used for the height and width dimension
- a `tuple` of two ints – in which case, the first *int* is used for the height dimension, and the second *int* for the width dimension

---

: Depending of the size of your kernel, several (of the last) columns of the input might be lost, because it is a valid cross-correlation, and not a full cross-correlation. It is up to the user to add proper padding.

- **in_channels** (*int*) – Number of channels in the input image
- **out_channels** (*int*) – Number of channels produced by the convolution
- **kernel_size** (*int or tuple*) – Size of the convolving kernel
- **stride** (*int or tuple, optional*) – Stride of the convolution
- **padding** (*int or tuple, optional*) – Zero-padding added to both sides of the input
- **dilation** (*int or tuple, optional*) – Spacing between kernel elements
- **groups** (*int, optional*) – Number of blocked connections from input channels to output channels
- **bias** (*bool, optional*) – If True, adds a learnable bias to the output

Shape:

- Input: $(N, C_{in}, H_{in}, W_{in})$
- Output: $(N, C_{out}, H_{out}, W_{out})$ where $H_{out} = floor((H_{in} + 2 * padding[0] - dilation[0] * (kernel\_size[0] - 1) - 1)/stride[0] + 1)$ $W_{out} = floor((W_{in} + 2 * padding[1] - dilation[1] * (kernel\_size[1] - 1) - 1)/stride[1] + 1)$

- **weight** (Tensor) – the learnable weights of the module of shape (out_channels, in_channels, kernel_size[0], kernel_size[1])
- **bias** (Tensor) – the learnable bias of the module of shape (out_channels)

Examples:

```
>>> # With square kernels and equal stride
>>> m = nn.Conv2d(16, 33, 3, stride=2)
>>> # non-square kernels and unequal stride and with padding
>>> m = nn.Conv2d(16, 33, (3, 5), stride=(2, 1), padding=(4, 2))
>>> # non-square kernels and unequal stride and with padding and dilation
>>> m = nn.Conv2d(16, 33, (3, 5), stride=(2, 1), padding=(4, 2), dilation=(3, 1))
>>> input = autograd.Variable(torch.randn(20, 16, 50, 100))
>>> output = m(input)
```

Conv3d

class `torch.nn.`**`Conv3d`**(*in_channels*, *out_channels*, *kernel_size*, *stride=1*, *padding=0*, *dilation=1*, *groups=1*, *bias=True*)

Applies a 3D convolution over an input signal composed of several input planes.

In the simplest case, the output value of the layer with input size (N, C_{in}, D, H, W) and output $(N, C_{out}, D_{out}, H_{out}, W_{out})$ can be precisely described as:

$$out(N_i, C_{out_j}) = bias(C_{out_j}) + \sum_{k=0}^{C_{in}-1} weight(C_{out_j}, k) \star input(N_i, k)$$

where $\star$ is the valid 3D cross-correlation operator

`stride` controls the stride for the cross-correlation.

If `padding` is non-zero, then the input is implicitly zero-padded on both sides for `padding` number of points

`dilation` controls the spacing between the kernel points. It is harder to describe, but this link has a nice visualization of what `dilation` does.

`groups` controls the connections between inputs and outputs.

At groups=1, all inputs are convolved to all outputs.

At groups=2, the operation becomes equivalent to having two conv layers side by side, each seeing half the input channels, and producing half the output channels, and both subsequently concatenated.

The parameters `kernel_size`, `stride`, `padding`, `dilation` can either be:

- a single `int` – in which case the same value is used for the height and width dimension
- a `tuple` of three ints – in which case, the first *int* is used for the depth dimension, the second *int* for the height dimension and the third *int* for the width dimension

: Depending of the size of your kernel, several (of the last) columns of the input might be lost, because it is a valid cross-correlation, and not a full cross-correlation. It is up to the user to add proper padding.

- **`in_channels`** (*`int`*) – Number of channels in the input image
- **`out_channels`** (*`int`*) – Number of channels produced by the convolution
- **`kernel_size`** (*`int or tuple`*) – Size of the convolving kernel
- **`stride`** (*`int or tuple, optional`*) – Stride of the convolution

- **padding** (*int or tuple, optional*) – Zero-padding added to both sides of the input
- **dilation** (*int or tuple, optional*) – Spacing between kernel elements
- **groups** (*int, optional*) – Number of blocked connections from input channels to output channels
- **bias** (*bool, optional*) – If True, adds a learnable bias to the output

Shape:

- Input: $(N, C_{in}, D_{in}, H_{in}, W_{in})$
- Output: $(N, C_{out}, D_{out}, H_{out}, W_{out})$ where $D_{out} = floor((D_{in} + 2 * padding[0] - dilation[0] * (kernel\_size[0] - 1) - 1)/stride[0] + 1)$ $H_{out} = floor((H_{in} + 2 * padding[1] - dilation[1] * (kernel\_size[1] - 1) - 1)/stride[1] + 1)$ $W_{out} = floor((W_{in} + 2 * padding[2] - dilation[2] * (kernel\_size[2] - 1) - 1)/stride[2] + 1)$

- **weight** (Tensor) – the learnable weights of the module of shape (out_channels, in_channels, kernel_size[0], kernel_size[1], kernel_size[2])
- **bias** (Tensor) – the learnable bias of the module of shape (out_channels)

Examples:

```
>>> # With square kernels and equal stride
>>> m = nn.Conv3d(16, 33, 3, stride=2)
>>> # non-square kernels and unequal stride and with padding
>>> m = nn.Conv3d(16, 33, (3, 5, 2), stride=(2, 1, 1), padding=(4, 2, 0))
>>> input = autograd.Variable(torch.randn(20, 16, 10, 50, 100))
>>> output = m(input)
```

ConvTranspose1d

class torch.nn.**ConvTranspose1d**(*in_channels*, *out_channels*, *kernel_size*, *stride=1*, *padding=0*, *output_padding=0*, *groups=1*, *bias=True*)

Applies a 1D transposed convolution operator over an input image composed of several input planes.

This module can be seen as the gradient of Conv1d with respect to its input. It is sometimes (but incorrectly) refered to as a deconvolutional operation.

: Depending of the size of your kernel, several (of the last) columns of the input might be lost, because it is a valid cross-correlation, and not a full cross-correlation. It is up to the user to add proper padding.

- **in_channels** (*int*) – Number of channels in the input image
- **out_channels** (*int*) – Number of channels produced by the convolution
- **kernel_size** (*int or tuple*) – Size of the convolving kernel
- **stride** (*int or tuple, optional*) – Stride of the convolution

- **padding** (*int or tuple, optional*) – Zero-padding added to both sides of the input
- **output_padding** (*int or tuple, optional*) – Zero-padding added to one side of the output
- **groups** (*int, optional*) – Number of blocked connections from input channels to output channels
- **bias** (*bool, optional*) – If True, adds a learnable bias to the output

Shape:

- Input: (N, C_{in}, L_{in})
- Output: (N, C_{out}, L_{out}) where $L_{out} = (L_{in} - 1) * stride - 2 * padding + kernel\_size + output\_padding$

- **weight** (Tensor) – the learnable weights of the module of shape (in_channels, out_channels, kernel_size[0], kernel_size[1])
- **bias** (Tensor) – the learnable bias of the module of shape (out_channels)

ConvTranspose2d

class torch.nn.**ConvTranspose2d**(*in_channels*, *out_channels*, *kernel_size*, *stride=1*, *padding=0*, *output_padding=0*, *groups=1*, *bias=True*)

Applies a 2D transposed convolution operator over an input image composed of several input planes.

This module can be seen as the gradient of Conv2d with respect to its input. It is sometimes (but incorrectly) refered to as a deconvolutional operation.

`stride` controls the stride for the cross-correlation.

If `padding` is non-zero, then the input is implicitly zero-padded on both sides for `padding` number of points

If `output_padding` is non-zero, then the output is implicitly zero-padded on one side for `output_padding` number of points

`dilation` controls the spacing between the kernel points. It is harder to describe, but this link has a nice visualization of what `dilation` does.

`groups` controls the connections between inputs and outputs.

At groups=1, all inputs are convolved to all outputs.

At groups=2, the operation becomes equivalent to having two conv layers side by side, each seeing half the input channels, and producing half the output channels, and both subsequently concatenated.

The parameters `kernel_size`, `stride`, `padding`, `output_padding` can either be:

- a single `int` – in which case the same value is used for the height and width dimension
- a `tuple` of two ints – in which case, the first *int* is used for the height dimension, and the second *int* for the width dimension

: Depending of the size of your kernel, several (of the last) columns of the input might be lost, because it is a valid cross-correlation, and not a full cross-correlation. It is up to the user to add proper padding.

- **in_channels** (*int*) – Number of channels in the input image
- **out_channels** (*int*) – Number of channels produced by the convolution
- **kernel_size** (*int or tuple*) – Size of the convolving kernel
- **stride** (*int or tuple, optional*) – Stride of the convolution
- **padding** (*int or tuple, optional*) – Zero-padding added to both sides of the input
- **output_padding** (*int or tuple, optional*) – Zero-padding added to one side of the output
- **groups** (*int, optional*) – Number of blocked connections from input channels to output channels
- **bias** (*bool, optional*) – If True, adds a learnable bias to the output

Shape:

- Input: $(N, C_{in}, H_{in}, W_{in})$
- Output: $(N, C_{out}, H_{out}, W_{out})$ where $H_{out} = (H_{in} - 1) * stride[0] - 2 * padding[0] + kernel\_size[0] + output\_padding[0]$ $W_{out} = (W_{in} - 1) * stride[1] - 2 * padding[1] + kernel\_size[1] + output\_padding[1]$

- **weight** (Tensor) – the learnable weights of the module of shape (in_channels, out_channels, kernel_size[0], kernel_size[1])
- **bias** (Tensor) – the learnable bias of the module of shape (out_channels)

Examples:

```
>>> # With square kernels and equal stride
>>> m = nn.ConvTranspose2d(16, 33, 3, stride=2)
>>> # non-square kernels and unequal stride and with padding
>>> m = nn.ConvTranspose2d(16, 33, (3, 5), stride=(2, 1), padding=(4, 2))
>>> input = autograd.Variable(torch.randn(20, 16, 50, 100))
>>> output = m(input)
>>> # exact output size can be also specified as an argument
>>> input = autograd.Variable(torch.randn(1, 16, 12, 12))
>>> downsample = nn.Conv2d(16, 16, 3, stride=2, padding=1)
>>> upsample = nn.ConvTranspose2d(16, 16, 3, stride=2, padding=1)
>>> h = downsample(input)
>>> h.size()
torch.Size([1, 16, 6, 6])
>>> output = upsample(h, output_size=input.size())
>>> output.size()
torch.Size([1, 16, 12, 12])
```

ConvTranspose3d

class torch.nn.**ConvTranspose3d**(*in_channels*, *out_channels*, *kernel_size*, *stride=1*, *padding=0*, *output_padding=0*, *groups=1*, *bias=True*)

Applies a 3D transposed convolution operator over an input image composed of several input planes. The

transposed convolution operator multiplies each input value element-wise by a learnable kernel, and sums over the outputs from all input feature planes.

This module can be seen as the exact reverse of Conv3d. It is sometimes (but incorrectly) refered to as a deconvolutional operation.

`stride` controls the stride for the cross-correlation.
If `padding` is non-zero, then the input is implicitly zero-padded on both sides for `padding` number of points
If `output_padding` is non-zero, then the output is implicitly zero-padded on one side for `output_padding` number of points
`groups` controls the connections between inputs and outputs.

At groups=1, all inputs are convolved to all outputs.

At groups=2, the operation becomes equivalent to having two conv layers side by side, each seeing half the input channels, and producing half the output channels, and both subsequently concatenated.

The parameters `kernel_size`, `stride`, `padding`, `output_padding` can either be:

- a single `int` – in which case the same value is used for the height and width dimension
- a `tuple` of three ints – in which case, the first *int* is used for the depth dimension, the second *int* for the width dimension and the third *int* for the width dimension

---

: Depending of the size of your kernel, several (of the last) columns of the input might be lost, because it is a valid cross-correlation, and not a full cross-correlation. It is up to the user to add proper padding.

- **in_channels** (*int*) – Number of channels in the input image
- **out_channels** (*int*) – Number of channels produced by the convolution
- **kernel_size** (*int or tuple*) – Size of the convolving kernel
- **stride** (*int or tuple, optional*) – Stride of the convolution
- **padding** (*int or tuple, optional*) – Zero-padding added to both sides of the input
- **output_padding** (*int or tuple, optional*) – Zero-padding added to one side of the output
- **groups** (*int, optional*) – Number of blocked connections from input channels to output channels
- **bias** (*bool, optional*) – If True, adds a learnable bias to the output

Shape:

- Input: $(N, C_{in}, D_{in}, H_{in}, W_{in})$
- Output: $(N, C_{out}, D_{out}, H_{out}, W_{out})$ where $D_{out} = (D_{in} - 1) * stride[0] - 2 * padding[0] + kernel\_size[0] + output\_padding[0]$ $H_{out} = (H_{in} - 1) * stride[1] - 2 * padding[1] + kernel\_size[1] + output\_padding[1]$ $W_{out} = (W_{in} - 1) * stride[2] - 2 * padding[2] + kernel\_size[2] + output\_padding[2]$

- **weight** (Tensor) – the learnable weights of the module of shape (in_channels, out_channels, kernel_size[0], kernel_size[1], kernel_size[2])
- **bias** (Tensor) – the learnable bias of the module of shape (out_channels)

Examples:

```
>>> # With square kernels and equal stride
>>> m = nn.ConvTranspose3d(16, 33, 3, stride=2)
>>> # non-square kernels and unequal stride and with padding
>>> m = nn.Conv3d(16, 33, (3, 5, 2), stride=(2, 1, 1), padding=(0, 4, 2))
>>> input = autograd.Variable(torch.randn(20, 16, 10, 50, 100))
>>> output = m(input)
```

Pooling Layers

MaxPool1d

class torch.nn.**MaxPool1d**(*kernel_size*, *stride=None*, *padding=0*, *dilation=1*, *return_indices=False*, *ceil_mode=False*)

Applies a 1D max pooling over an input signal composed of several input planes.

In the simplest case, the output value of the layer with input size (N, C, L) and output (N, C, L_{out}) can be precisely described as:

$$out(N_i, C_j, k) = \max_{m=0}^{kernel\_size-1} input(N_i, C_j, stride * k + m)$$

If `padding` is non-zero, then the input is implicitly zero-padded on both sides for `padding` number of points `dilation` controls the spacing between the kernel points. It is harder to describe, but this link has a nice visualization of what `dilation` does.

- **kernel_size** – the size of the window to take a max over
- **stride** – the stride of the window. Default value is `kernel_size`
- **padding** – implicit zero padding to be added on both sides
- **dilation** – a parameter that controls the stride of elements in the window
- **return_indices** – if True, will return the max indices along with the outputs. Useful when Unpooling later
- **ceil_mode** – when True, will use *ceil* instead of *floor* to compute the output shape

Shape:

- Input: (N, C, L_{in})
- Output: (N, C, L_{out}) where $L_{out} = floor((L_{in} + 2 * padding - dilation * (kernel\_size - 1) - 1)/stride + 1)$

Examples:

```
>>> # pool of size=3, stride=2
>>> m = nn.MaxPool1d(3, stride=2)
>>> input = autograd.Variable(torch.randn(20, 16, 50))
>>> output = m(input)
```

MaxPool2d

class `torch.nn.`**`MaxPool2d`**(*kernel_size*, *stride=None*, *padding=0*, *dilation=1*, *return_indices=False*, *ceil_mode=False*)

Applies a 2D max pooling over an input signal composed of several input planes.

In the simplest case, the output value of the layer with input size (N, C, H, W), output (N, C, H_{out}, W_{out}) and `kernel_size` (kH, kW) can be precisely described as:

$$out(N_i, C_j, h, w) = \max_{m=0}^{kH-1} \max_{n=0}^{kW-1} input(N_i, C_j, stride[0] * h + m, stride[1] * w + n)$$

If `padding` is non-zero, then the input is implicitly zero-padded on both sides for `padding` number of points `dilation` controls the spacing between the kernel points. It is harder to describe, but this link has a nice visualization of what `dilation` does.

The parameters `kernel_size`, `stride`, `padding`, `dilation` can either be:

- a single `int` – in which case the same value is used for the height and width dimension
- a `tuple` of two ints – in which case, the first *int* is used for the height dimension, and the second *int* for the width dimension

- **`kernel_size`** – the size of the window to take a max over
- **`stride`** – the stride of the window. Default value is `kernel_size`
- **`padding`** – implicit zero padding to be added on both sides
- **`dilation`** – a parameter that controls the stride of elements in the window
- **`return_indices`** – if True, will return the max indices along with the outputs. Useful when Unpooling later
- **`ceil_mode`** – when True, will use *ceil* instead of *floor* to compute the output shape

Shape:

- Input: (N, C, H_{in}, W_{in})
- Output: (N, C, H_{out}, W_{out}) where $H_{out} = floor((H_{in} + 2 * padding[0] - dilation[0] * (kernel\_size[0] - 1) - 1)/stride[0] + 1)$ $W_{out} = floor((W_{in} + 2 * padding[1] - dilation[1] * (kernel\_size[1] - 1) - 1)/stride[1] + 1)$

Examples:

```
>>> # pool of square window of size=3, stride=2
>>> m = nn.MaxPool2d(3, stride=2)
>>> # pool of non-square window
>>> m = nn.MaxPool2d((3, 2), stride=(2, 1))
>>> input = autograd.Variable(torch.randn(20, 16, 50, 32))
>>> output = m(input)
```

MaxPool3d

class `torch.nn.`**`MaxPool3d`**(*kernel_size*, *stride=None*, *padding=0*, *dilation=1*, *return_indices=False*, *ceil_mode=False*)

Applies a 3D max pooling over an input signal composed of several input planes.

In the simplest case, the output value of the layer with input size (N, C, D, H, W), output $(N, C, D_{out}, H_{out}, W_{out})$ and `kernel_size` (kD, kH, kW) can be precisely described as:

$$out(N_i, C_j, d, h, w) = \max_{k=0}^{kD-1} \max_{m=0}^{kH-1} \max_{n=0}^{kW-1} input(N_i, C_j, stride[0] * k + d, stride[1] * h + m, stride[2] * w + n)$$

If `padding` is non-zero, then the input is implicitly zero-padded on both sides for `padding` number of points `dilation` controls the spacing between the kernel points. It is harder to describe, but this link has a nice visualization of what `dilation` does.

The parameters `kernel_size`, `stride`, `padding`, `dilation` can either be:

- a single `int` – in which case the same value is used for the height and width dimension
- a `tuple` of three ints – in which case, the first *int* is used for the depth dimension, the second *int* for the width dimension and the third *int* for the width dimension

- **`kernel_size`** – the size of the window to take a max over
- **`stride`** – the stride of the window. Default value is `kernel_size`
- **`padding`** – implicit zero padding to be added on both sides
- **`dilation`** – a parameter that controls the stride of elements in the window
- **`return_indices`** – if True, will return the max indices along with the outputs. Useful when Unpooling later
- **`ceil_mode`** – when True, will use *ceil* instead of *floor* to compute the output shape

Shape:

- Input: $(N, C, D_{in}, H_{in}, W_{in})$
- Output: $(N, C, D_{out}, H_{out}, W_{out})$ where $D_{out} = floor((D_{in} + 2 * padding[0] - dilation[0] * (kernel\_size[0] - 1) - 1)/stride[0] + 1)$ $H_{out} = floor((H_{in} + 2 * padding[1] - dilation[1] * (kernel\_size[1] - 1) - 1)/stride[1] + 1)$ $W_{out} = floor((W_{in} + 2 * padding[2] - dilation[2] * (kernel\_size[2] - 1) - 1)/stride[2] + 1)$

Examples:

```
>>> # pool of square window of size=3, stride=2
>>> m = nn.MaxPool3d(3, stride=2)
>>> # pool of non-square window
>>> m = nn.MaxPool3d((3, 2, 2), stride=(2, 1, 2))
>>> input = autograd.Variable(torch.randn(20, 16, 50,44, 31))
>>> output = m(input)
```

MaxUnpool1d

class `torch.nn.`**`MaxUnpool1d`**(*kernel_size*, *stride=None*, *padding=0*)

Computes a partial inverse of *MaxPool1d*.

MaxPool1d is not fully invertible, since the non-maximal values are lost.

MaxUnpool1d takes in as input the output of *MaxPool1d* including the indices of the maximal values and computes a partial inverse in which all non-maximal values are set to zero.

: *MaxPool1d* can map several input sizes to the same output sizes. Hence, the inversion process can get ambiguous. To accommodate this, you can provide the needed output size as an additional argument *output_size* in the forward call. See the Inputs and Example below.

- **`kernel_size`** (*int or tuple*) – Size of the max pooling window.
- **`stride`** (*int or tuple*) – Stride of the max pooling window. It is set to `kernel_size` by default.
- **`padding`** (*int or tuple*) – Padding that was added to the input

Inputs:

- *input*: the input Tensor to invert
- *indices*: the indices given out by *MaxPool1d*
- *output_size* (optional) : a *torch.Size* that specifies the targeted output size

Shape:

- Input: (N, C, H_{in})
- Output: (N, C, H_{out}) where $H_{out} = (H_{in} - 1) * stride[0] - 2 * padding[0] + kernel\_size[0]$ or as given by `output_size` in the call operator

Example:

```
>>> pool = nn.MaxPool1d(2, stride=2, return_indices=True)
>>> unpool = nn.MaxUnpool1d(2, stride=2)
>>> input = Variable(torch.Tensor([[[1, 2, 3, 4, 5, 6, 7, 8]]]))
>>> output, indices = pool(input)
>>> unpool(output, indices)
Variable containing:
(0 ,.,.) =
   0   2   0   4   0   6   0   8
[torch.FloatTensor of size 1x1x8]

>>> # Example showcasing the use of output_size
>>> input = Variable(torch.Tensor([[[1, 2, 3, 4, 5, 6, 7, 8, 9]]]))
>>> output, indices = pool(input)
>>> unpool(output, indices, output_size=input.size())
Variable containing:
(0 ,.,.) =
   0   2   0   4   0   6   0   8   0
[torch.FloatTensor of size 1x1x9]
```

```
>>> unpool(output, indices)
Variable containing:
(0 ,.,.) =
   0   2   0   4   0   6   0   8
[torch.FloatTensor of size 1x1x8]
```

MaxUnpool2d

class `torch.nn.`**`MaxUnpool2d`**(*kernel_size*, *stride=None*, *padding=0*)

Computes a partial inverse of *MaxPool2d*.

MaxPool2d is not fully invertible, since the non-maximal values are lost.

MaxUnpool2d takes in as input the output of *MaxPool2d* including the indices of the maximal values and computes a partial inverse in which all non-maximal values are set to zero.

: *MaxPool2d* can map several input sizes to the same output sizes. Hence, the inversion process can get ambiguous. To accommodate this, you can provide the needed output size as an additional argument *output_size* in the forward call. See the Inputs and Example below.

- **`kernel_size`** (*int or tuple*) – Size of the max pooling window.
- **`stride`** (*int or tuple*) – Stride of the max pooling window. It is set to `kernel_size` by default.
- **`padding`** (*int or tuple*) – Padding that was added to the input

Inputs:

- *input*: the input Tensor to invert
- *indices*: the indices given out by *MaxPool2d*
- *output_size* (optional) : a *torch.Size* that specifies the targeted output size

Shape:

- Input: (N, C, H_{in}, W_{in})
- Output: (N, C, H_{out}, W_{out}) where $H_{out} = (H_{in} - 1) * stride[0] - 2 * padding[0] + kernel\_size[0]$ $W_{out} = (W_{in} - 1) * stride[1] - 2 * padding[1] + kernel\_size[1]$ or as given by `output_size` in the call operator

Example:

```
>>> pool = nn.MaxPool2d(2, stride=2, return_indices=True)
>>> unpool = nn.MaxUnpool2d(2, stride=2)
>>> input = Variable(torch.Tensor([[[[ 1,  2,  3,  4],
...                                  [ 5,  6,  7,  8],
...                                  [ 9, 10, 11, 12],
...                                  [13, 14, 15, 16]]]]))
>>> output, indices = pool(input)
>>> unpool(output, indices)
Variable containing:
(0 ,0 ,.,.) =
```

```
   0   0   0   0
   0   6   0   8
   0   0   0   0
   0  14   0  16
[torch.FloatTensor of size 1x1x4x4]

>>> # specify a different output size than input size
>>> unpool(output, indices, output_size=torch.Size([1, 1, 5, 5]))
Variable containing:
(0 ,0 ,.,.) =
   0   0   0   0   0
   6   0   8   0   0
   0   0   0  14   0
  16   0   0   0   0
   0   0   0   0   0
[torch.FloatTensor of size 1x1x5x5]
```

MaxUnpool3d

class `torch.nn.`**`MaxUnpool3d`**(*kernel_size*, *stride=None*, *padding=0*)

Computes a partial inverse of *MaxPool3d*.

MaxPool3d is not fully invertible, since the non-maximal values are lost. *MaxUnpool3d* takes in as input the output of *MaxPool3d* including the indices of the maximal values and computes a partial inverse in which all non-maximal values are set to zero.

: *MaxPool3d* can map several input sizes to the same output sizes. Hence, the inversion process can get ambiguous. To accommodate this, you can provide the needed output size as an additional argument *output_size* in the forward call. See the Inputs section below.

- **`kernel_size`** (*int or tuple*) – Size of the max pooling window.
- **`stride`** (*int or tuple*) – Stride of the max pooling window. It is set to `kernel_size` by default.
- **`padding`** (*int or tuple*) – Padding that was added to the input

Inputs:

- *input*: the input Tensor to invert
- *indices*: the indices given out by *MaxPool3d*
- *output_size* (optional) : a *torch.Size* that specifies the targeted output size

Shape:

- Input: $(N, C, D_{in}, H_{in}, W_{in})$
- Output: $(N, C, D_{out}, H_{out}, W_{out})$ where $D_{out} = (D_{in} - 1) * stride[0] - 2 * padding[0] + kernel\_size[0]$ $H_{out} = (H_{in} - 1) * stride[1] - 2 * padding[1] + kernel\_size[1]$ $W_{out} = (W_{in} - 1) * stride[2] - 2 * padding[2] + kernel\_size[2]$ or as given by `output_size` in the call operator

Example:

```
>>> # pool of square window of size=3, stride=2
>>> pool = nn.MaxPool3d(3, stride=2, return_indices=True)
>>> unpool = nn.MaxUnpool3d(3, stride=2)
>>> output, indices = pool(Variable(torch.randn(20, 16, 51, 33, 15)))
>>> unpooled_output = unpool(output, indices)
>>> unpooled_output.size()
torch.Size([20, 16, 51, 33, 15])
```

AvgPool1d

class `torch.nn.`**`AvgPool1d`**(*kernel_size*, *stride=None*, *padding=0*, *ceil_mode=False*, *count_include_pad=True*)

Applies a 1D average pooling over an input signal composed of several input planes.

In the simplest case, the output value of the layer with input size (N, C, L), output (N, C, L_{out}) and `kernel_size` k can be precisely described as:

$$out(N_i, C_j, l) = 1/k * \sum_{m=0}^{k} input(N_i, C_j, stride * l + m)$$

If `padding` is non-zero, then the input is implicitly zero-padded on both sides for `padding` number of points

The parameters `kernel_size`, `stride`, `padding` can each be an `int` or a one-element tuple.

- **`kernel_size`** – the size of the window
- **`stride`** – the stride of the window. Default value is `kernel_size`
- **`padding`** – implicit zero padding to be added on both sides
- **`ceil_mode`** – when True, will use *ceil* instead of *floor* to compute the output shape
- **`count_include_pad`** – when True, will include the zero-padding in the averaging calculation

Shape:

- Input: (N, C, L_{in})
- Output: (N, C, L_{out}) where $L_{out} = floor((L_{in} + 2 * padding - kernel\_size)/stride + 1)$

Examples:

```
>>> # pool with window of size=3, stride=2
>>> m = nn.AvgPool1d(3, stride=2)
>>> m(Variable(torch.Tensor([[[1,2,3,4,5,6,7]]])))
Variable containing:
(0 ,.,.) =
  2  4  6
[torch.FloatTensor of size 1x1x3]
```

AvgPool2d

class `torch.nn.`**`AvgPool2d`**(*kernel_size*, *stride=None*, *padding=0*, *ceil_mode=False*, *count_include_pad=True*)

Applies a 2D average pooling over an input signal composed of several input planes.

In the simplest case, the output value of the layer with input size (N, C, H, W), output (N, C, H_{out}, W_{out}) and `kernel_size` (kH, kW) can be precisely described as:

$$out(N_i, C_j, h, w) = 1/(kH * kW) * \sum_{m=0}^{kH-1} \sum_{n=0}^{kW-1} input(N_i, C_j, stride[0] * h + m, stride[1] * w + n)$$

If `padding` is non-zero, then the input is implicitly zero-padded on both sides for `padding` number of points

The parameters `kernel_size`, `stride`, `padding` can either be:

- a single `int` – in which case the same value is used for the height and width dimension
- a `tuple` of two ints – in which case, the first *int* is used for the height dimension, and the second *int* for the width dimension

- **`kernel_size`** – the size of the window
- **`stride`** – the stride of the window. Default value is `kernel_size`
- **`padding`** – implicit zero padding to be added on both sides
- **`ceil_mode`** – when True, will use *ceil* instead of *floor* to compute the output shape
- **`count_include_pad`** – when True, will include the zero-padding in the averaging calculation

Shape:

- Input: (N, C, H_{in}, W_{in})
- Output: (N, C, H_{out}, W_{out}) where $H_{out} = floor((H_{in} + 2 * padding[0] - kernel\_size[0])/stride[0]+1)$ $W_{out} = floor((W_{in}+2*padding[1]-kernel\_size[1])/stride[1]+1)$

Examples:

```
>>> # pool of square window of size=3, stride=2
>>> m = nn.AvgPool2d(3, stride=2)
>>> # pool of non-square window
>>> m = nn.AvgPool2d((3, 2), stride=(2, 1))
>>> input = autograd.Variable(torch.randn(20, 16, 50, 32))
>>> output = m(input)
```

AvgPool3d

class `torch.nn.`**`AvgPool3d`**(*kernel_size*, *stride=None*)

Applies a 3D average pooling over an input signal composed of several input planes.

In the simplest case, the output value of the layer with input size (N, C, D, H, W), output $(N, C, D_{out}, H_{out}, W_{out})$ and `kernel_size` (kD, kH, kW) can be precisely described as:

$$out(N_i, C_j, d, h, w) = 1/(kD * kH * kW) * \sum_{k=0}^{kD-1} \sum_{m=0}^{kH-1} \sum_{n=0}^{kW-1} input(N_i, C_j, stride[0] * d + k, stride[1] * h + m, str$$

The parameters `kernel_size`, `stride` can either be:

- a single `int` – in which case the same value is used for the height and width dimension
- a `tuple` of three ints – in which case, the first *int* is used for the depth dimension, the second *int* for the width dimension and the third *int* for the width dimension

- **kernel_size** – the size of the window
- **stride** – the stride of the window. Default value is `kernel_size`

Shape:

- Input: $(N, C, D_{in}, H_{in}, W_{in})$
- Output: $(N, C, D_{out}, H_{out}, W_{out})$ where $D_{out} = floor((D_{in} - kernel\_size[0])/stride[0] + 1)$ $H_{out} = floor((H_{in} - kernel\_size[1])/stride[1] + 1)$ $W_{out} = floor((W_{in} - kernel\_size[2])/stride[2] + 1)$

Examples:

```
>>> # pool of square window of size=3, stride=2
>>> m = nn.AvgPool3d(3, stride=2)
>>> # pool of non-square window
>>> m = nn.AvgPool3d((3, 2, 2), stride=(2, 1, 2))
>>> input = autograd.Variable(torch.randn(20, 16, 50,44, 31))
>>> output = m(input)
```

FractionalMaxPool2d

class `torch.nn.`**`FractionalMaxPool2d`**(*kernel_size*, *output_size=None*, *output_ratio=None*, *return_indices=False*, *_random_samples=None*)

Applies a 2D fractional max pooling over an input signal composed of several input planes.

Fractiona MaxPooling is described in detail in the paper Fractional MaxPooling by Ben Graham

The max-pooling operation is applied in kHxkW 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.

- **kernel_size** – the size of the window to take a max over. Can be a single number k (for a square kernel of k x k) or a tuple (kh x kw)
- **output_size** – the target output size of the image of the form oH x oW. Can be a tuple (oH, oW) or a single number oH for a square image oH x oH
- **output_ratio** – If one wants to have an output size as a ratio of the input size, this option can be given. This has to be a number or tuple in the range (0, 1)
- **return_indices** – if True, will return the indices along with the outputs. Useful to pass to nn.MaxUnpool2d . Default: False

Examples

```
>>> # pool of square window of size=3, and target output size 13x12
>>> m = nn.FractionalMaxPool2d(3, output_size=(13, 12))
>>> # pool of square window and target output size being half of input image size
>>> m = nn.FractionalMaxPool2d(3, output_ratio=(0.5, 0.5))
>>> input = autograd.Variable(torch.randn(20, 16, 50, 32))
>>> output = m(input)
```

LPPool2d

class `torch.nn.`**`LPPool2d`**(*norm_type*, *kernel_size*, *stride=None*, *ceil_mode=False*)

Applies a 2D power-average pooling over an input signal composed of several input planes.

On each window, the function computed is: $f(X) = pow(sum(pow(X, p)), 1/p)$

- At p = infinity, one gets Max Pooling
- At p = 1, one gets Average Pooling

The parameters `kernel_size`, `stride` can either be:

- a single `int` – in which case the same value is used for the height and width dimension
- a `tuple` of two ints – in which case, the first *int* is used for the height dimension, and the second *int* for the width dimension

- **`kernel_size`** – the size of the window
- **`stride`** – the stride of the window. Default value is `kernel_size`
- **`ceil_mode`** – when True, will use *ceil* instead of *floor* to compute the output shape

Shape:

- Input: (N, C, H_{in}, W_{in})
- Output: (N, C, H_{out}, W_{out}) where $H_{out} = floor((H_{in} + 2 * padding[0] - dilation[0] * (kernel\_size[0] - 1) - 1)/stride[0] + 1)$ $W_{out} = floor((W_{in} + 2 * padding[1] - dilation[1] * (kernel\_size[1] - 1) - 1)/stride[1] + 1)$

Examples:

```
>>> # power-2 pool of square window of size=3, stride=2
>>> m = nn.LPPool2d(2, 3, stride=2)
>>> # pool of non-square window of power 1.2
>>> m = nn.LPPool2d(1.2, (3, 2), stride=(2, 1))
>>> input = autograd.Variable(torch.randn(20, 16, 50, 32))
>>> output = m(input)
```

AdaptiveMaxPool1d

class `torch.nn.`**`AdaptiveMaxPool1d`**(*output_size*, *return_indices=False*)

Applies a 1D adaptive max pooling over an input signal composed of several input planes.

The output size is H, for any input size. The number of output features is equal to the number of input planes.

- **output_size** – the target output size H
- **return_indices** – if True, will return the indices along with the outputs. Useful to pass to nn.MaxUnpool2d . Default: False

Examples

```
>>> # target output size of 5
>>> m = nn.AdaptiveMaxPool1d(5)
>>> input = autograd.Variable(torch.randn(1, 64, 8))
>>> output = m(input)
```

AdaptiveMaxPool2d

class torch.nn.**AdaptiveMaxPool2d**(*output_size*, *return_indices=False*)

Applies a 2D adaptive max pooling over an input signal composed of several input planes.

The output is of size H x W, for any input size. The number of output features is equal to the number of input planes.

- **output_size** – the target output size of the image of the form H x W. Can be a tuple (H, W) or a single number H for a square image H x H
- **return_indices** – if True, will return the indices along with the outputs. Useful to pass to nn.MaxUnpool2d . Default: False

Examples

```
>>> # target output size of 5x7
>>> m = nn.AdaptiveMaxPool2d((5,7))
>>> input = autograd.Variable(torch.randn(1, 64, 8, 9))
>>> # target output size of 7x7 (square)
>>> m = nn.AdaptiveMaxPool2d(7)
>>> input = autograd.Variable(torch.randn(1, 64, 10, 9))
>>> output = m(input)
```

AdaptiveAvgPool1d

class torch.nn.**AdaptiveAvgPool1d**(*output_size*)

Applies a 1D adaptive average pooling over an input signal composed of several input planes.

The output size is H, for any input size. The number of output features is equal to the number of input planes.

output_size – the target output size H

Examples

```
>>> # target output size of 5
>>> m = nn.AdaptiveAvgPool1d(5)
>>> input = autograd.Variable(torch.randn(1, 64, 8))
>>> output = m(input)
```

AdaptiveAvgPool2d

class `torch.nn.`**`AdaptiveAvgPool2d`**(*output_size*)

Applies a 2D adaptive average pooling over an input signal composed of several input planes.

The output is of size H x W, for any input size. The number of output features is equal to the number of input planes.

> **`output_size`** – the target output size of the image of the form H x W. Can be a tuple (H, W) or a single number H for a square image H x H

Examples

```
>>> # target output size of 5x7
>>> m = nn.AdaptiveAvgPool2d((5,7))
>>> input = autograd.Variable(torch.randn(1, 64, 8, 9))
>>> # target output size of 7x7 (square)
>>> m = nn.AdaptiveAvgPool2d(7)
>>> input = autograd.Variable(torch.randn(1, 64, 10, 9))
>>> output = m(input)
```

Non-linear Activations

ReLU

class `torch.nn.`**`ReLU`**(*inplace=False*)

Applies the rectified linear unit function element-wise $ReLU(x) = max(0, x)$

> **`inplace`** – can optionally do the operation in-place

Shape:

- Input: $(N, *)$ where * means, any number of additional dimensions
- Output: $(N, *)$, same shape as the input

Examples:

```
>>> m = nn.ReLU()
>>> input = autograd.Variable(torch.randn(2))
>>> print(input)
>>> print(m(input))
```

ReLU6

class torch.nn.**ReLU6**(*inplace=False*)
Applies the element-wise function $ReLU6(x) = min(max(0, x), 6)$

inplace – can optionally do the operation in-place

Shape:

- Input: $(N, *)$ where * means, any number of additional dimensions
- Output: $(N, *)$, same shape as the input

Examples:

```
>>> m = nn.ReLU6()
>>> input = autograd.Variable(torch.randn(2))
>>> print(input)
>>> print(m(input))
```

ELU

class torch.nn.**ELU**(*alpha=1.0*, *inplace=False*)
Applies element-wise, $f(x) = max(0, x) + min(0, alpha * (exp(x) - 1))$

- **alpha** – the alpha value for the ELU formulation
- **inplace** – can optionally do the operation in-place

Shape:

- Input: $(N, *)$ where * means, any number of additional dimensions
- Output: $(N, *)$, same shape as the input

Examples:

```
>>> m = nn.ELU()
>>> input = autograd.Variable(torch.randn(2))
>>> print(input)
>>> print(m(input))
```

PReLU

class torch.nn.**PReLU**(*num_parameters=1*, *init=0.25*)
Applies element-wise the function $PReLU(x) = max(0, x) + a * min(0, x)$ Here "a" is a learnable parameter. When called without arguments, nn.PReLU() uses a single parameter "a" across all input channels. If called with nn.PReLU(nChannels), a separate "a" is used for each input channel.

: weight decay should not be used when learning "a" for good performance.

- **num_parameters** – number of "a" to learn. Default: 1
- **init** – the initial value of "a". Default: 0.25

Shape:

- Input: $(N, *)$ where * means, any number of additional dimensions
- Output: $(N, *)$, same shape as the input

Examples:

```
>>> m = nn.PReLU()
>>> input = autograd.Variable(torch.randn(2))
>>> print(input)
>>> print(m(input))
```

LeakyReLU

class `torch.nn.`**LeakyReLU**(*negative_slope=0.01*, *inplace=False*)

Applies element-wise, $f(x) = max(0, x) + negative\_slope * min(0, x)$

- **negative_slope** – Controls the angle of the negative slope. Default: 1e-2
- **inplace** – can optionally do the operation in-place

Shape:

- Input: $(N, *)$ where * means, any number of additional dimensions
- Output: $(N, *)$, same shape as the input

Examples:

```
>>> m = nn.LeakyReLU(0.1)
>>> input = autograd.Variable(torch.randn(2))
>>> print(input)
>>> print(m(input))
```

Threshold

class `torch.nn.`**Threshold**(*threshold*, *value*, *inplace=False*)

Thresholds each element of the input Tensor

Threshold is defined as:

```
y =  x        if x >= threshold
     value    if x <  threshold
```

- **threshold** – The value to threshold at
- **value** – The value to replace with
- **inplace** – can optionally do the operation in-place

Shape:

- Input: $(N, *)$ where * means, any number of additional dimensions
- Output: $(N, *)$, same shape as the input

Examples:

```
>>> m = nn.Threshold(0.1, 20)
>>> input = Variable(torch.randn(2))
>>> print(input)
>>> print(m(input))
```

Hardtanh

class `torch.nn.`**`Hardtanh`**(*min_value=-1*, *max_value=1*, *inplace=False*)

Applies the HardTanh function element-wise

HardTanh is defined as:

```
f(x) = +1, if x  >  1
f(x) = -1, if x  < -1
f(x) =  x,  otherwise
```

The range of the linear region $[-1, 1]$ can be adjusted

- **`min_value`** – minimum value of the linear region range
- **`max_value`** – maximum value of the linear region range
- **`inplace`** – can optionally do the operation in-place

Shape:

- Input: $(N, *)$ where * means, any number of additional dimensions
- Output: $(N, *)$, same shape as the input

Examples:

```
>>> m = nn.HardTanh(-2, 2)
>>> input = autograd.Variable(torch.randn(2))
>>> print(input)
>>> print(m(input))
```

Sigmoid

class `torch.nn.`**`Sigmoid`**

Applies the element-wise function $f(x) = 1/(1 + exp(-x))$

Shape:

- Input: $(N, *)$ where * means, any number of additional dimensions
- Output: $(N, *)$, same shape as the input

Examples:

```
>>> m = nn.Sigmoid()
>>> input = autograd.Variable(torch.randn(2))
>>> print(input)
>>> print(m(input))
```

Tanh

class `torch.nn.`**`Tanh`**

Applies element-wise, $f(x) = (exp(x) - exp(-x))/(exp(x) + exp(-x))$

Shape:

- Input: $(N, *)$ where * means, any number of additional dimensions
- Output: $(N, *)$, same shape as the input

Examples:

```
>>> m = nn.Tanh()
>>> input = autograd.Variable(torch.randn(2))
>>> print(input)
>>> print(m(input))
```

LogSigmoid

class `torch.nn.`**`LogSigmoid`**

Applies element-wise $LogSigmoid(x) = log(1/(1 + exp(-x_i)))$

Shape:

- Input: $(N, *)$ where * means, any number of additional dimensions
- Output: $(N, *)$, same shape as the input

Examples:

```
>>> m = nn.LogSigmoid()
>>> input = autograd.Variable(torch.randn(2))
>>> print(input)
>>> print(m(input))
```

Softplus

class `torch.nn.`**`Softplus`**(*beta=1*, *threshold=20*)

Applies element-wise $f(x) = 1/beta * log(1 + exp(beta * x_i))$

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.

- **`beta`** – the beta value for the Softplus formulation. Default: 1
- **`threshold`** – values above this revert to a linear function. Default: 20

Shape:

- Input: $(N, *)$ where * means, any number of additional dimensions
- Output: $(N, *)$, same shape as the input

Examples:

```
>>> m = nn.Softplus()
>>> input = autograd.Variable(torch.randn(2))
>>> print(input)
>>> print(m(input))
```

Softshrink

class `torch.nn.`**`Softshrink`**(*lambd=0.5*)

Applies the soft shrinkage function elementwise

SoftShrinkage operator is defined as:

```
f(x) = x-lambda, if x > lambda >  f(x) = x+lambda, if x < -lambda
f(x) = 0, otherwise
```

`lambd` – the lambda value for the Softshrink formulation. Default: 0.5

Shape:

- Input: $(N, *)$ where * means, any number of additional dimensions
- Output: $(N, *)$, same shape as the input

Examples:

```
>>> m = nn.Softshrink()
>>> input = autograd.Variable(torch.randn(2))
>>> print(input)
>>> print(m(input))
```

Softsign

class `torch.nn.`**`Softsign`**

Applies element-wise, the function $f(x) = x/(1 + |x|)$

Shape:

- Input: $(N, *)$ where * means, any number of additional dimensions
- Output: $(N, *)$, same shape as the input

Examples:

```
>>> m = nn.Softsign()
>>> input = autograd.Variable(torch.randn(2))
>>> print(input)
>>> print(m(input))
```

Tanhshrink

class torch.nn.**Tanhshrink**

Applies element-wise, $Tanhshrink(x) = x - Tanh(x)$

Shape:

- Input: $(N, *)$ where * means, any number of additional dimensions
- Output: $(N, *)$, same shape as the input

Examples:

```
>>> m = nn.Tanhshrink()
>>> input = autograd.Variable(torch.randn(2))
>>> print(input)
>>> print(m(input))
```

Softmin

class torch.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

$f(x) = exp(-x_i - shift)/sum_j exp(-x_j - shift)$

where $shift = max_i - x_i$

Shape:

- Input: (N, L)
- Output: (N, L)

a Tensor of the same dimension and shape as the input, with values in the range [0, 1]

Examples:

```
>>> m = nn.Softmin()
>>> input = autograd.Variable(torch.randn(2, 3))
>>> print(input)
>>> print(m(input))
```

Softmax

class torch.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*

Shape:

- Input: (N, L)
- Output: (N, L)

a Tensor of the same dimension and shape as the input with values in the range [0, 1]

: This module doesn't work directly with NLLLoss, which expects the Log to be computed between the Softmax and itself. Use Logsoftmax instead (it's faster).

Examples:

```
>>> m = nn.Softmax()
>>> input = autograd.Variable(torch.randn(2, 3))
>>> print(input)
>>> print(m(input))
```

LogSoftmax

class `torch.nn.`**`LogSoftmax`**

Applies the Log(Softmax(x)) function to an n-dimensional input Tensor. The LogSoftmax formulation can be simplified as

$f_i(x) = log(1/a * exp(x_i))$ where $a = sum_j exp(x_j)$

Shape:

- Input: (N, L)
- Output: (N, L)

a Tensor of the same dimension and shape as the input with values in the range [-inf, 0)

Examples:

```
>>> m = nn.LogSoftmax()
>>> input = autograd.Variable(torch.randn(2, 3))
>>> print(input)
>>> print(m(input))
```

Normalization layers

BatchNorm1d

class `torch.nn.`**`BatchNorm1d`**(*num_features*, *eps=1e-05*, *momentum=0.1*, *affine=True*)

Applies Batch Normalization over a 2d or 3d input that is seen as a mini-batch.

$$y = \frac{x - mean[x]}{\sqrt{Var[x] + \epsilon}} * gamma + beta$$

The mean and standard-deviation are calculated per-dimension over the mini-batches and gamma and beta are learnable parameter vectors of size N (where N is the input size).

During training, this layer keeps a running estimate of its computed mean and variance. The running sum is kept with a default momentum of 0.1.

During evaluation, this running mean/variance is used for normalization.

- **num_features** – num_features from an expected input of size *batch_size x num_features [x width]*
- **eps** – a value added to the denominator for numerical stability. Default: 1e-5
- **momentum** – the value used for the running_mean and running_var computation. Default: 0.1
- **affine** – a boolean value that when set to true, gives the layer learnable affine parameters.

Shape:

- Input: (N, C) or (N, C, L)
- Output: (N, C) or (N, C, L) (same shape as input)

Examples

```
>>> # With Learnable Parameters
>>> m = nn.BatchNorm1d(100)
>>> # Without Learnable Parameters
>>> m = nn.BatchNorm1d(100, affine=False)
>>> input = autograd.Variable(torch.randn(20, 100))
>>> output = m(input)
```

BatchNorm2d

class `torch.nn.`**BatchNorm2d**(*num_features*, *eps=1e-05*, *momentum=0.1*, *affine=True*)

Applies Batch Normalization over a 4d input that is seen as a mini-batch of 3d inputs

$$y = \frac{x - mean[x]}{\sqrt{Var[x] + \epsilon}} * gamma + beta$$

The mean and standard-deviation are calculated per-dimension over the mini-batches and gamma and beta are learnable parameter vectors of size N (where N is the input size).

During training, this layer keeps a running estimate of its computed mean and variance. The running sum is kept with a default momentum of 0.1.

During evaluation, this running mean/variance is used for normalization.

- **num_features** – num_features from an expected input of size batch_size x num_features x height x width
- **eps** – a value added to the denominator for numerical stability. Default: 1e-5
- **momentum** – the value used for the running_mean and running_var computation. Default: 0.1
- **affine** – a boolean value that when set to true, gives the layer learnable affine parameters.

Shape:

- Input: (N, C, H, W)
- Output: (N, C, H, W) (same shape as input)

Examples

```
>>> # With Learnable Parameters
>>> m = nn.BatchNorm2d(100)
>>> # Without Learnable Parameters
>>> m = nn.BatchNorm2d(100, affine=False)
>>> input = autograd.Variable(torch.randn(20, 100, 35, 45))
>>> output = m(input)
```

BatchNorm3d

class `torch.nn.`**`BatchNorm3d`**(*num_features*, *eps=1e-05*, *momentum=0.1*, *affine=True*)

Applies Batch Normalization over a 5d input that is seen as a mini-batch of 4d inputs

$$y = \frac{x - mean[x]}{\sqrt{Var[x] + \epsilon}} * gamma + beta$$

The mean and standard-deviation are calculated per-dimension over the mini-batches and gamma and beta are learnable parameter vectors of size N (where N is the input size).

During training, this layer keeps a running estimate of its computed mean and variance. The running sum is kept with a default momentum of 0.1.

During evaluation, this running mean/variance is used for normalization.

- **`num_features`** – num_features from an expected input of size batch_size x num_features x height x width
- **`eps`** – a value added to the denominator for numerical stability. Default: 1e-5
- **`momentum`** – the value used for the running_mean and running_var computation. Default: 0.1
- **`affine`** – a boolean value that when set to true, gives the layer learnable affine parameters.

Shape:

- Input: (N, C, D, H, W)
- Output: (N, C, D, H, W) (same shape as input)

Examples

```
>>> # With Learnable Parameters
>>> m = nn.BatchNorm3d(100)
>>> # Without Learnable Parameters
>>> m = nn.BatchNorm3d(100, affine=False)
>>> input = autograd.Variable(torch.randn(20, 100, 35, 45, 10))
>>> output = m(input)
```

Recurrent layers

RNN

class torch.nn.**RNN**(**args*, ***kwargs*)

Applies a multi-layer Elman RNN with tanh or ReLU non-linearity to an input sequence.

For each element in the input sequence, each layer computes the following function:

$$h_t = \tanh(w_{ih} * x_t + b_{ih} + w_{hh} * h_{(t-1)} + b_{hh})$$

where h_t is the hidden state at time *t*, and x_t is the hidden state of the previous layer at time *t* or $input_t$ for the first layer. If nonlinearity='relu', then *ReLU* is used instead of *tanh*.

- **input_size** – The number of expected features in the input x
- **hidden_size** – The number of features in the hidden state h
- **num_layers** – Number of recurrent layers.
- **nonlinearity** – The non-linearity to use ['tanh'|'relu']. Default: 'tanh'
- **bias** – If False, then the layer does not use bias weights b_ih and b_hh. Default: True
- **batch_first** – If True, then the input and output tensors are provided as (batch, seq, feature)
- **dropout** – If non-zero, introduces a dropout layer on the outputs of each RNN layer except the last layer
- **bidirectional** – If True, becomes a bidirectional RNN. Default: False

Inputs: input, h_0

- **input** (seq_len, batch, input_size): tensor containing the features of the input sequence. The input can also be a packed variable length sequence. See *torch.nn.utils.rnn.pack_padded_sequence()* for details.
- **h_0** (num_layers * num_directions, batch, hidden_size): tensor containing the initial hidden state for each element in the batch.

Outputs: output, h_n

- **output** (seq_len, batch, hidden_size * num_directions): tensor containing the output features (h_k) from the last layer of the RNN, for each k. If a *torch.nn.utils.rnn.PackedSequence* has been given as the input, the output will also be a packed sequence.
- **h_n** (num_layers * num_directions, batch, hidden_size): tensor containing the hidden state for k=seq_len.

- **weight_ih_l[k]** – the learnable input-hidden weights of the k-th layer, of shape *(input_size x hidden_size)*
- **weight_hh_l[k]** – the learnable hidden-hidden weights of the k-th layer, of shape *(hidden_size x hidden_size)*
- **bias_ih_l[k]** – the learnable input-hidden bias of the k-th layer, of shape *(hidden_size)*

- **bias_hh_l[k]** – the learnable hidden-hidden bias of the k-th layer, of shape *(hidden_size)*

Examples:

```
>>> rnn = nn.RNN(10, 20, 2)
>>> input = Variable(torch.randn(5, 3, 10))
>>> h0 = Variable(torch.randn(2, 3, 20))
>>> output, hn = rnn(input, h0)
```

LSTM

class `torch.nn.`**`LSTM`**`(*args, **kwargs)`

Applies a multi-layer long short-term memory (LSTM) RNN to an input sequence.

For each element in the input sequence, each layer computes the following function:

$$
\begin{aligned}
i_t &= sigmoid(W_{ii}x_t + b_{ii} + W_{hi}h_{(t-1)} + b_{hi}) \\
f_t &= sigmoid(W_{if}x_t + b_{if} + W_{hf}h_{(t-1)} + b_{hf}) \\
g_t &= \tanh(W_{ig}x_t + b_{ig} + W_{hc}h_{(t-1)} + b_{hg}) \\
o_t &= sigmoid(W_{io}x_t + b_{io} + W_{ho}h_{(t-1)} + b_{ho}) \\
c_t &= f_t * c_{(t-1)} + i_t * g_t \\
h_t &= o_t * \tanh(c_t)
\end{aligned}
$$

where h_t is the hidden state at time *t*, c_t is the cell state at time *t*, x_t is the hidden state of the previous layer at time *t* or $input_t$ for the first layer, and i_t, f_t, g_t, o_t are the input, forget, cell, and out gates, respectively.

- **input_size** – The number of expected features in the input x
- **hidden_size** – The number of features in the hidden state h
- **num_layers** – Number of recurrent layers.
- **bias** – If False, then the layer does not use bias weights b_ih and b_hh. Default: True
- **batch_first** – If True, then the input and output tensors are provided as (batch, seq, feature)
- **dropout** – If non-zero, introduces a dropout layer on the outputs of each RNN layer except the last layer
- **bidirectional** – If True, becomes a bidirectional RNN. Default: False

Inputs: input, (h_0, c_0)

- **input** (seq_len, batch, input_size): tensor containing the features of the input sequence. The input can also be a packed variable length sequence. See *`torch.nn.utils.rnn.pack_padded_sequence()`* for details.
- **h_0** (num_layers * num_directions, batch, hidden_size): tensor containing the initial hidden state for each element in the batch.
- **c_0** (num_layers * num_directions, batch, hidden_size): tensor containing the initial cell state for each element in the batch.

Outputs: output, (h_n, c_n)

- **output** (seq_len, batch, hidden_size * num_directions): tensor containing the output features *(h_t)* from the last layer of the RNN, for each t. If a `torch.nn.utils.rnn.PackedSequence` has been given as the input, the output will also be a packed sequence.
- **h_n** (num_layers * num_directions, batch, hidden_size): tensor containing the hidden state for t=seq_len
- **c_n** (num_layers * num_directions, batch, hidden_size): tensor containing the cell state for t=seq_len

- **weight_ih_l[k]** – the learnable input-hidden weights of the k-th layer *(W_ii|W_if|W_ig|W_io)*, of shape *(input_size x 4*hidden_size)*
- **weight_hh_l[k]** – the learnable hidden-hidden weights of the k-th layer *(W_hi|W_hf|W_hg|W_ho)*, of shape *(hidden_size x 4*hidden_size)*
- **bias_ih_l[k]** – the learnable input-hidden bias of the k-th layer *(b_ii|b_if|b_ig|b_io)*, of shape *(4*hidden_size)*
- **bias_hh_l[k]** – the learnable hidden-hidden bias of the k-th layer *(W_hi|W_hf|W_hg|b_ho)*, of shape *(4*hidden_size)*

Examples:

```
>>> rnn = nn.LSTM(10, 20, 2)
>>> input = Variable(torch.randn(5, 3, 10))
>>> h0 = Variable(torch.randn(2, 3, 20))
>>> c0 = Variable(torch.randn(2, 3, 20))
>>> output, hn = rnn(input, (h0, c0))
```

GRU

class `torch.nn.`**`GRU`**`(*args, **kwargs)`

Applies a multi-layer gated recurrent unit (GRU) RNN to an input sequence.

For each element in the input sequence, each layer computes the following function:

$$
\begin{aligned}
r_t &= sigmoid(W_{ir}x_t + b_{ir} + W_{hr}h_{(t-1)} + b_{hr}) \\
i_t &= sigmoid(W_{ii}x_t + b_{ii} + W_h ih_{(t-1)} + b_{hi}) \\
n_t &= \tanh(W_{in}x_t + b_{in} + r_t * (W_{hn}h_{(t-1)} + b_{hn})) \\
h_t &= (1 - i_t) * n_t + i_t * h_{(t-1)}
\end{aligned}
$$

where h_t is the hidden state at time *t*, x_t is the hidden state of the previous layer at time *t* or $input_t$ for the first layer, and r_t, i_t, n_t are the reset, input, and new gates, respectively.

- **input_size** – The number of expected features in the input x
- **hidden_size** – The number of features in the hidden state h
- **num_layers** – Number of recurrent layers.
- **bias** – If False, then the layer does not use bias weights b_ih and b_hh. Default: True
- **batch_first** – If True, then the input and output tensors are provided as (batch, seq, feature)
- **dropout** – If non-zero, introduces a dropout layer on the outputs of each RNN layer except the last layer

- **bidirectional** – If True, becomes a bidirectional RNN. Default: False

Inputs: input, h_0

- **input** (seq_len, batch, input_size): tensor containing the features of the input sequence. The input can also be a packed variable length sequence. See *torch.nn.utils.rnn.pack_padded_sequence()* for details.
- **h_0** (num_layers * num_directions, batch, hidden_size): tensor containing the initial hidden state for each element in the batch.

Outputs: output, h_n

- **output** (seq_len, batch, hidden_size * num_directions): tensor containing the output features h_t from the last layer of the RNN, for each t. If a *torch.nn.utils.rnn.PackedSequence* has been given as the input, the output will also be a packed sequence.
- **h_n** (num_layers * num_directions, batch, hidden_size): tensor containing the hidden state for t=seq_len

- **weight_ih_l[k]** – the learnable input-hidden weights of the k-th layer (W_ir|W_ii|W_in), of shape *(input_size x 3*hidden_size)*
- **weight_hh_l[k]** – the learnable hidden-hidden weights of the k-th layer (W_hr|W_hi|W_hn), of shape *(hidden_size x 3*hidden_size)*
- **bias_ih_l[k]** – the learnable input-hidden bias of the k-th layer (b_ir|b_ii|b_in), of shape *(3*hidden_size)*
- **bias_hh_l[k]** – the learnable hidden-hidden bias of the k-th layer (W_hr|W_hi|W_hn), of shape *(3*hidden_size)*

Examples:

```
>>> rnn = nn.GRU(10, 20, 2)
>>> input = Variable(torch.randn(5, 3, 10))
>>> h0 = Variable(torch.randn(2, 3, 20))
>>> output, hn = rnn(input, h0)
```

RNNCell

class torch.nn.**RNNCell**(*input_size*, *hidden_size*, *bias=True*, *nonlinearity='tanh'*)

An Elman RNN cell with tanh or ReLU non-linearity.

$$h' = \tanh(w_{ih} * x + b_{ih} + w_{hh} * h + b_{hh})$$

If nonlinearity='relu', then ReLU is used in place of tanh.

- **input_size** – The number of expected features in the input x
- **hidden_size** – The number of features in the hidden state h
- **bias** – If False, then the layer does not use bias weights b_ih and b_hh. Default: True
- **nonlinearity** – The non-linearity to use ['tanh'|'relu']. Default: 'tanh'

Inputs: input, hidden

- **input** (batch, input_size): tensor containing input features
- **hidden** (batch, hidden_size): tensor containing the initial hidden state for each element in the batch.

Outputs: h'

- **h'** (batch, hidden_size): tensor containing the next hidden state for each element in the batch

- **weight_ih** – the learnable input-hidden weights, of shape *(input_size x hidden_size)*
- **weight_hh** – the learnable hidden-hidden weights, of shape *(hidden_size x hidden_size)*
- **bias_ih** – the learnable input-hidden bias, of shape *(hidden_size)*
- **bias_hh** – the learnable hidden-hidden bias, of shape *(hidden_size)*

Examples:

```
>>> rnn = nn.RNNCell(10, 20)
>>> input = Variable(torch.randn(6, 3, 10))
>>> hx = Variable(torch.randn(3, 20))
>>> output = []
>>> for i in range(6):
...     hx = rnn(input[i], hx)
...     output.append(hx)
```

LSTMCell

class `torch.nn.`**`LSTMCell`**(*input_size*, *hidden_size*, *bias=True*)

A long short-term memory (LSTM) cell.

$$
\begin{aligned}
i &= sigmoid(W_{ii}x + b_{ii} + W_{hi}h + b_{hi}) \\
f &= sigmoid(W_{if}x + b_{if} + W_{hf}h + b_{hf}) \\
g &= \tanh(W_{ig}x + b_{ig} + W_{hc}h + b_{hg}) \\
o &= sigmoid(W_{io}x + b_{io} + W_{ho}h + b_{ho}) \\
c' &= f * c + i * g \\
h' &= o * \tanh(c_t)
\end{aligned}
$$

- **input_size** – The number of expected features in the input x
- **hidden_size** – The number of features in the hidden state h
- **bias** – If *False*, then the layer does not use bias weights *b_ih* and *b_hh*. Default: True

Inputs: input, (h_0, c_0)

- **input** (batch, input_size): tensor containing input features
- **h_0** (batch, hidden_size): tensor containing the initial hidden state for each element in the batch.
- **c_0** (batch. hidden_size): tensor containing the initial cell state for each element in the batch.

Outputs: h_1, c_1

- **h_1** (batch, hidden_size): tensor containing the next hidden state for each element in the batch
- **c_1** (batch, hidden_size): tensor containing the next cell state for each element in the batch

- **weight_ih** – the learnable input-hidden weights, of shape *(input_size x hidden_size)*
- **weight_hh** – the learnable hidden-hidden weights, of shape *(hidden_size x hidden_size)*
- **bias_ih** – the learnable input-hidden bias, of shape *(hidden_size)*
- **bias_hh** – the learnable hidden-hidden bias, of shape *(hidden_size)*

Examples:

```
>>> rnn = nn.LSTMCell(10, 20)
>>> input = Variable(torch.randn(6, 3, 10))
>>> hx = Variable(torch.randn(3, 20))
>>> cx = Variable(torch.randn(3, 20))
>>> output = []
>>> for i in range(6):
...     hx, cx = rnn(input[i], (hx, cx))
...     output.append(hx)
```

GRUCell

class `torch.nn.`**`GRUCell`**(*input_size*, *hidden_size*, *bias=True*)

A gated recurrent unit (GRU) cell

$$
\begin{aligned}
r &= sigmoid(W_{ir}x + b_{ir} + W_{hr}h + b_{hr}) \\
i &= sigmoid(W_{ii}x + b_{ii} + W_{hi}h + b_{hi}) \\
n &= \tanh(W_{in}x + b_{in} + r * (W_{hn}h + b_{hn})) \\
h' &= (1 - i) * n + i * h
\end{aligned}
$$

- **input_size** – The number of expected features in the input x
- **hidden_size** – The number of features in the hidden state h
- **bias** – If *False*, then the layer does not use bias weights *b_ih* and *b_hh*. Default: *True*

Inputs: input, hidden

- **input** (batch, input_size): tensor containing input features
- **hidden** (batch, hidden_size): tensor containing the initial hidden state for each element in the batch.

Outputs: h'

- **h'**: (batch, hidden_size): tensor containing the next hidden state for each element in the batch

- **weight_ih** – the learnable input-hidden weights, of shape *(input_size x hidden_size)*
- **weight_hh** – the learnable hidden-hidden weights, of shape *(hidden_size x hidden_size)*
- **bias_ih** – the learnable input-hidden bias, of shape *(hidden_size)*
- **bias_hh** – the learnable hidden-hidden bias, of shape *(hidden_size)*

Examples:

```
>>> rnn = nn.GRUCell(10, 20)
>>> input = Variable(torch.randn(6, 3, 10))
>>> hx = Variable(torch.randn(3, 20))
>>> output = []
>>> for i in range(6):
...     hx = rnn(input[i], hx)
...     output.append(hx)
```

Linear layers

Linear

class `torch.nn.`**`Linear`**(*in_features*, *out_features*, *bias=True*)

Applies a linear transformation to the incoming data: $y = Ax + b$

- **`in_features`** – size of each input sample
- **`out_features`** – size of each output sample
- **`bias`** – If set to False, the layer will not learn an additive bias. Default: True

Shape:

- Input: $(N, in\_features)$
- Output: $(N, out\_features)$

- **`weight`** – the learnable weights of the module of shape (out_features x in_features)
- **`bias`** – the learnable bias of the module of shape (out_features)

Examples:

```
>>> m = nn.Linear(20, 30)
>>> input = autograd.Variable(torch.randn(128, 20))
>>> output = m(input)
>>> print(output.size())
```

Dropout layers

Dropout

class `torch.nn.`**`Dropout`**(*p=0.5*, *inplace=False*)

Randomly zeroes some of the elements of the input tensor. The elements to zero are randomized on every forward call.

- **`p`** – probability of an element to be zeroed. Default: 0.5
- **`inplace`** – If set to True, will do this operation in-place. Default: false

Shape:

- Input: *Any*. Input can be of any shape
- Output: *Same*. Output is of the same shape as input

Examples:

```
>>> m = nn.Dropout(p=0.2)
>>> input = autograd.Variable(torch.randn(20, 16))
>>> output = m(input)
```

Dropout2d

class `torch.nn.`**`Dropout2d`**(*p=0.5*, *inplace=False*)

Randomly zeroes whole channels of the input tensor. The channels to zero-out are randomized on every forward call.

Usually the input comes from Conv2d modules.

As described in the paper Efficient Object Localization Using Convolutional Networks , 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.Dropout2d()` will help promote independence between feature maps and should be used instead.

- **p** (*float, optional*) – probability of an element to be zeroed.
- **inplace** (*bool, optional*) – If set to True, will do this operation in-place

Shape:

- Input: (N, C, H, W)
- Output: (N, C, H, W) (same shape as input)

Examples:

```
>>> m = nn.Dropout2d(p=0.2)
>>> input = autograd.Variable(torch.randn(20, 16, 32, 32))
>>> output = m(input)
```

Dropout3d

class `torch.nn.`**`Dropout3d`**(*p=0.5*, *inplace=False*)

Randomly zeroes whole channels of the input tensor. The channels to zero are randomized on every forward call.

Usually the input comes from Conv3d modules.

As described in the paper Efficient Object Localization Using Convolutional Networks , 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.Dropout3d()` will help promote independence between feature maps and should be used instead.

- **p** (*float, optional*) – probability of an element to be zeroed.
- **inplace** (*bool, optional*) – If set to True, will do this operation in-place

Shape:

- Input: (N, C, D, H, W)
- Output: (N, C, D, H, W) (same shape as input)

Examples:

```
>>> m = nn.Dropout3d(p=0.2)
>>> input = autograd.Variable(torch.randn(20, 16, 4, 32, 32))
>>> output = m(input)
```

Sparse layers

Embedding

class `torch.nn.`**Embedding**(*num_embeddings*, *embedding_dim*, *padding_idx=None*, *max_norm=None*, *norm_type=2*, *scale_grad_by_freq=False*, *sparse=False*)

A simple lookup table that stores embeddings of a fixed dictionary and size.

This module is often used to store word embeddings and retrieve them using indices. The input to the module is a list of indices, and the output is the corresponding word embeddings.

- **num_embeddings** (*int*) – size of the dictionary of embeddings
- **embedding_dim** (*int*) – the size of each embedding vector
- **padding_idx** (*int, optional*) – If given, pads the output with zeros whenever it encounters the index.
- **max_norm** (*float, optional*) – If given, will renormalize the embeddings to always have a norm lesser than this
- **norm_type** (*float, optional*) – The p of the p-norm to compute for the max_norm option
- **scale_grad_by_freq** (*boolean, optional*) – if given, this will scale gradients by the frequency of the words in the dictionary.

weight (Tensor) – the learnable weights of the module of shape (num_embeddings, embedding_dim)

Shape:

- Input: LongTensor *(N, W)*, N = mini-batch, W = number of indices to extract per mini-batch
- Output: *(N, W, embedding_dim)*

Examples:

```
>>> # an Embedding module containing 10 tensors of size 3
>>> embedding = nn.Embedding(10, 3)
>>> # a batch of 2 samples of 4 indices each
>>> input = Variable(torch.LongTensor([[1,2,4,5],[4,3,2,9]]))
>>> embedding(input)

Variable containing:
(0 ,.,.) =
 -1.0822  1.2522  0.2434
  0.8393 -0.6062 -0.3348
  0.6597  0.0350  0.0837
  0.5521  0.9447  0.0498

(1 ,.,.) =
  0.6597  0.0350  0.0837
 -0.1527  0.0877  0.4260
  0.8393 -0.6062 -0.3348
 -0.8738 -0.9054  0.4281
[torch.FloatTensor of size 2x4x3]

>>> # example with padding_idx
>>> embedding = nn.Embedding(10, 3, padding_idx=0)
>>> input = Variable(torch.LongTensor([[0,2,0,5]]))
>>> embedding(input)

Variable containing:
(0 ,.,.) =
  0.0000  0.0000  0.0000
  0.3452  0.4937 -0.9361
  0.0000  0.0000  0.0000
  0.0706 -2.1962 -0.6276
[torch.FloatTensor of size 1x4x3]
```

Distance functions

PairwiseDistance

class `torch.nn.`**`PairwiseDistance`**(*p=2*, *eps=1e-06*)

Computes the batchwise pairwise distance between vectors v1,v2:

$$\|x\|_p := \left(\sum_{i=1}^{n} |x_i|^p \right)^{1/p}$$

Args: x (Tensor): input tensor containing the two input batches p (real): the norm degree. Default: 2

Shape:

- Input: (N, D) where *D = vector dimension*
- Output: $(N, 1)$

```
>>> pdist = nn.PairwiseDistance(2)
>>> input1 = autograd.Variable(torch.randn(100, 128))
```

```
>>> input2 = autograd.Variable(torch.randn(100, 128))
>>> output = pdist(input1, input2)
```

Loss functions

L1Loss

class `torch.nn.`**`L1Loss`**(*size_average=True*)

Creates a criterion that measures the mean absolute value of the element-wise difference between input *x* and target *y*:

$loss(x, y) = 1/n \sum |x_i - y_i|$

x and *y* arbitrary shapes with a total of *n* elements each.

The sum operation still operates over all the elements, and divides by *n*.

The division by *n* can be avoided if one sets the constructor argument *sizeAverage=False*

MSELoss

class `torch.nn.`**`MSELoss`**(*size_average=True*)

Creates a criterion that measures the mean squared error between *n* elements in the input *x* and target *y*:

$loss(x, y) = 1/n \sum |x_i - y_i|^2$

x and *y* arbitrary shapes with a total of *n* elements each.

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*.

CrossEntropyLoss

class `torch.nn.`**`CrossEntropyLoss`**(*weight=None*, *size_average=True*)

This criterion combines *LogSoftMax* and *NLLLoss* in one single class.

It is useful when training 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* is expected to contain scores for each class.

input has to be a 2D *Tensor* of size *batch x n*.

This criterion expects a class index (0 to nClasses-1) as the *target* for each value of a 1D tensor of size *n*

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])))
```

The losses are averaged across observations for each minibatch.

Shape:

- Input: (N, C) where *C = number of classes*
- Target: (N) where each value is *0 <= targets[i] <= C-1*

NLLLoss

class `torch.nn.`**`NLLLoss`**(*weight=None*, *size_average=True*)

The negative log likelihood loss. 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 call is expected to contain log-probabilities of each class: input has to be a 2D Tensor of size *(minibatch, n)*

Obtaining log-probabilities in a neural network is easily achieved by adding a *LogSoftmax* layer in the last layer of your network.

You may use *CrossEntropyLoss* instead, if you prefer not to add an extra layer.

The target that this loss expects is a class index *(0 to N-1, where N = number of classes)*

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]
```

- **`weight`** (*Tensor, optional*) – a manual rescaling weight given to each class. If given, has to be a Tensor of size "nclasses"
- **`size_average`** (*bool, optional*) – 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.

Shape:

- Input: (N, C) where *C = number of classes*
- Target: (N) where each value is *0 <= targets[i] <= C-1*

`weight` – the class-weights given as input to the constructor

Examples:

```
>>> m = nn.LogSoftmax()
>>> loss = nn.NLLLoss()
>>> # input is of size nBatch x nClasses = 3 x 5
>>> input = autograd.Variable(torch.randn(3, 5), requires_grad=True)
>>> # each element in target has to have 0 <= value < nclasses
>>> target = autograd.Variable(torch.LongTensor([1, 0, 4]))
```

```
>>> output = loss(m(input), target)
>>> output.backward()
```

NLLLoss2d

class `torch.nn.`**`NLLLoss2d`**(*weight=None*, *size_average=True*)

This is negative log likehood loss, but for image inputs. It computes NLL loss per-pixel.

- **`weight`** (`Tensor, optional`) – a manual rescaling weight given to each class. If given, has to be a 1D Tensor having as many elements, as there are classes.
- **`size_average`** – 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. Default: True

Shape:

- Input: (N, C, H, W) where *C = number of classes*
- Target: (N, H, W) where each value is *0 <= targets[i] <= C-1*

Examples

```
>>> m = nn.Conv2d(16, 32, (3, 3)).float()
>>> loss = nn.NLLLoss2d()
>>> # input is of size nBatch x nClasses x height x width
>>> input = autograd.Variable(torch.randn(3, 16, 10, 10))
>>> # each element in target has to have 0 <= value < nclasses
>>> target = autograd.Variable(torch.LongTensor(3, 8, 8).random_(0, 4))
>>> output = loss(m(input), target)
>>> output.backward()
```

KLDivLoss

class `torch.nn.`**`KLDivLoss`**(*weight=None*, *size_average=True*)

The Kullback-Leibler divergence Loss

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 *NLLLoss*, the *input* given is expected to contain *log-probabilities*, however unlike *ClassNLLLoss*, *input* is not restricted to a 2D Tensor, because the criterion is applied element-wise.

This criterion expects a *target Tensor* of the same size as the *input Tensor*.

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.

BCELoss

class torch.nn.**BCELoss**(*weight=None*, *size_average=True*)

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 targets *t[i]* should be numbers between 0 and 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.

MarginRankingLoss

class torch.nn.**MarginRankingLoss**(*margin=0*, *size_average=True*)

Creates a criterion that measures the loss given inputs *x1*, *x2*, two 1D min-batch *Tensor's, and a label 1D mini-batch tensor 'y* with values (*1* or *-1*).

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 for each sample in the mini-batch is:

```
loss(x, y) = max(0, -y * (x1 - x2) + margin)
```

if the internal variable *sizeAverage* = *True*, the loss function averages the loss over the batch samples; if *sizeAverage* = *False*, then the loss function sums over the batch samples. By default, *sizeAverage* equals to *True*.

HingeEmbeddingLoss

class torch.nn.**HingeEmbeddingLoss**(*size_average=True*)

Measures the loss given an input *x* which is a 2D mini-batch tensor and a labels *y*, a 1D tensor containg values (*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
```

x and *y* arbitrary shapes with a total of *n* elements each 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=False*.

The *margin* has a default value of *1*, or can be set in the constructor.

MultiLabelMarginLoss

class torch.nn.**MultiLabelMarginLoss**(*size_average=True*)

Creates a criterion that optimizes a multi-class multi-classification hinge loss (margin-based loss) between input *x* (a 2D mini-batch *Tensor*) and output *y* (which is a 2D *Tensor* of target class indices). For each sample in the mini-batch:

```
loss(x, y) = sum_ij(max(0, 1 - (x[y[j]] - x[i]))) / x.size(0)
```

where *i == 0* to *x.size(0)*, *j == 0* to *y.size(0)*, *y[j] != 0*, and *i != y[j]* for all *i* and *j*.

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

SmoothL1Loss

class torch.nn.**SmoothL1Loss**(*size_average=True*)

Creates a criterion that uses a squared term if the absolute element-wise error falls below 1 and an L1 term otherwise. It is less sensitive to outliers than the *MSELoss* and in some cases prevents exploding gradients (e.g. see "Fast R-CNN" paper by Ross Girshick). Also known as the Huber loss:

```
                       { 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
```

x and *y* arbitrary shapes with a total of *n* elements each 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*

SoftMarginLoss

class torch.nn.**SoftMarginLoss**(*size_average=True*)

Creates a criterion that optimizes a two-class classification logistic loss between input *x* (a 2D mini-batch Tensor) and target *y* (which is a tensor containing either *1* or *-1*).

```
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*.

MultiLabelSoftMarginLoss

class torch.nn.**MultiLabelSoftMarginLoss**(*weight=None*, *size_average=True*)

Creates a criterion that optimizes a multi-label one-versus-all loss based on max-entropy, between input *x* (a 2D mini-batch *Tensor*) and target *y* (a binary 2D *Tensor*). For each sample in the minibatch:

```
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 == 0* to *x.nElement()-1*, *y[i] in {0,1}*. *y* and *x* must have the same size.

CosineEmbeddingLoss

class torch.nn.**CosineEmbeddingLoss**(*margin=0*, *size_average=True*)

Creates a criterion that measures the loss given an input tensors x1, x2 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. 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
```

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* = *True*.

MultiMarginLoss

class torch.nn.**MultiMarginLoss**(*p=1*, *margin=1*, *weight=None*, *size_average=True*)

Creates a criterion that optimizes a multi-class classification hinge loss (margin-based loss) between input *x* (a 2D mini-batch *Tensor*) and output *y* (which is a 1D tensor of target class indices, *0* <= *y* <= *x.size(1)*):

For each mini-batch sample:

```
loss(x, y) = sum_i(max(0, (margin - x[y] + x[i]))^p) / x.size(0)
             where `i == 0` to `x.size(0)` and `i != y`.
```

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(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.

Vision layers

PixelShuffle

class torch.nn.**PixelShuffle**(*upscale_factor*)

Rearranges elements in a Tensor of shape $(*, C * r^2, H, W]$ to a tensor of shape $(C, H * r, W * r)$.

This is useful for implementing efficient sub-pixel convolution with a stride of $1/r$.

Look at the paper: Real-Time Single Image and Video Super-Resolution Using an Efficient Sub-Pixel Convolutional Neural Network by Shi et. al (2016) for more details

upscale_factor (*int*) – factor to increase spatial resolution by

Shape:

- Input: $(N, C * upscale\_factor^2, H, W)$
- Output: $(N, C, H * upscale\_factor, W * upscale\_factor)$

Examples:

```
>>> ps = nn.PixelShuffle(3)
>>> input = autograd.Variable(torch.Tensor(1, 9, 4, 4))
>>> output = ps(input)
>>> print(output.size())
torch.Size([1, 1, 12, 12])
```

UpsamplingNearest2d

class `torch.nn.`**`UpsamplingNearest2d`**(*size=None*, *scale_factor=None*)

Applies a 2D nearest neighbor upsampling to an input signal composed of several input channels.

To specify the scale, it takes either the `size` or the `scale_factor` as it's constructor argument.

When *size* is given, it is the output size of the image (h, w).

- **`size`** (*`tuple, optional`*) – a tuple of ints (H_out, W_out) output sizes
- **`scale_factor`** (*`int, optional`*) – the multiplier for the image height / width

Shape:

- Input: (N, C, H_{in}, W_{in})
- Output: (N, C, H_{out}, W_{out}) where $H_{out} = floor(H_{in} * scale\_factor)$ $W_{out} = floor(W_{in} * scale\_factor)$

Examples:

```
>>> inp
Variable containing:
(0 ,0 ,.,.) =
  1  2
  3  4
[torch.FloatTensor of size 1x1x2x2]

>>> m = nn.UpsamplingNearest2d(scale_factor=2)
>>> m(inp)
Variable containing:
(0 ,0 ,.,.) =
  1  1  2  2
  1  1  2  2
  3  3  4  4
  3  3  4  4
[torch.FloatTensor of size 1x1x4x4]
```

UpsamplingBilinear2d

class `torch.nn.`**`UpsamplingBilinear2d`**(*size=None*, *scale_factor=None*)

Applies a 2D bilinear upsampling to an input signal composed of several input channels.

To specify the scale, it takes either the `size` or the `scale_factor` as it's constructor argument.

When *size* is given, it is the output size of the image (h, w).

- **size** (*tuple, optional*) – a tuple of ints (H_out, W_out) output sizes
- **scale_factor** (*int, optional*) – the multiplier for the image height / width

Shape:

- Input: (N, C, H_{in}, W_{in})
- Output: (N, C, H_{out}, W_{out}) where $H_{out} = floor(H_{in} * scale\_factor)$ $W_{out} = floor(W_{in} * scale\_factor)$

Examples:

```
>>> inp
Variable containing:
(0 ,0 ,.,.) =
  1  2
  3  4
[torch.FloatTensor of size 1x1x2x2]

>>> m = nn.UpsamplingBilinear2d(scale_factor=2)
>>> m(inp)
Variable containing:
(0 ,0 ,.,.) =
  1.0000  1.3333  1.6667  2.0000
  1.6667  2.0000  2.3333  2.6667
  2.3333  2.6667  3.0000  3.3333
  3.0000  3.3333  3.6667  4.0000
[torch.FloatTensor of size 1x1x4x4]
```

Multi-GPU layers

DataParallel

class torch.nn.**DataParallel**(*module*, *device_ids=None*, *output_device=None*, *dim=0*)

Implements data parallelism at the module level.

This container parallelizes the application of the given module by splitting the input across the specified devices by chunking in the batch dimension. In the forward pass, the module is replicated on each device, and each replica handles a portion of the input. During the backwards pass, gradients from each replica are summed into the original module.

The batch size should be larger than the number of GPUs used. It should also be an integer multiple of the number of GPUs so that each chunk is the same size (so that each GPU processes the same number of samples).

See also: *Use nn.DataParallel instead of multiprocessing*

Arbitrary positional and keyword inputs are allowed to be passed into DataParallel EXCEPT Tensors. All variables will be scattered on dim specified (default 0). Primitive types will be broadcasted, but all other types will be a shallow copy and can be corrupted if written to in the model's forward pass.

- **module** – module to be parallelized
- **device_ids** – CUDA devices (default: all devices)

- **output_device** – device location of output (default: device_ids[0])

Example:

```
>>> net = torch.nn.DataParallel(model, device_ids=[0, 1, 2])
>>> output = net(input_var)
```

Utilities

clip_grad_norm

torch.nn.utils.**clip_grad_norm**(*parameters*, *max_norm*, *norm_type=2*)

Clips gradient norm of an iterable of parameters.

The norm is computed over all gradients together, as if they were concatenated into a single vector. Gradients are modified in-place.

- **parameters** (*Iterable[Variable]*) – an iterable of Variables that will have gradients normalized
- **max_norm** (*float or int*) – max norm of the gradients
- **norm_type** (*float or int*) – type of the used p-norm. Can be `'inf'` for infinity norm.

Total norm of the parameters (viewed as a single vector).

PackedSequence

torch.nn.utils.rnn.**PackedSequence**(*_cls*, *data*, *batch_sizes*)

Holds the data and list of batch_sizes of a packed sequence.

All RNN modules accept packed sequences as inputs.

: Instances of this class should never be created manually. They are meant to be instantiated by functions like *pack_padded_sequence()*.

- **data** (Variable) – Variable containing packed sequence
- **batch_sizes** (*list[int]*) – list of integers holding information about the batch size at each sequence step

pack_padded_sequence

torch.nn.utils.rnn.**pack_padded_sequence**(*input*, *lengths*, *batch_first=False*)

Packs a Variable containing padded sequences of variable length.

Input can be of size `TxBx*` where T is the length of the longest sequence (equal to `lengths[0]`), B is the batch size, and * is any number of dimensions (including 0). If `batch_first` is True `BxTx*` inputs are expected.

The sequences should be sorted by length in a decreasing order, i.e. `input[:,0]` should be the longest sequence, and `input[:,B-1]` the shortest one.

: This function accept any input that has at least two dimensions. You can apply it to pack the labels, and use the output of the RNN with them to compute the loss directly. A Variable can be retrieved from a *PackedSequence* object by accessing its `.data` attribute.

- **input** (Variable) – padded batch of variable length sequences.
- **lengths** (*list[int]*) – list of sequences lengths of each batch element.
- **batch_first** (*bool, optional*) – if True, the input is expected in BxTx* format.

a *PackedSequence* object

pad_packed_sequence

torch.nn.utils.rnn.**pad_packed_sequence**(*sequence*, *batch_first=False*)

Pads a packed batch of variable length sequences.

It is an inverse operation to *pack_padded_sequence()*.

The returned Variable's data will be of size TxBx*, where T is the length of the longest sequence and B is the batch size. If `batch_size` is True, the data will be transposed into BxTx* format.

Batch elements will be ordered decreasingly by their length.

- **sequence** (PackedSequence) – batch to pad
- **batch_first** (*bool, optional*) – if True, the output will be in BxTx* format.

Tuple of Variable containing the padded sequence, and a list of lengths of each sequence in the batch.

CHAPTER 10

torch.nn.functional

Convolution functions

conv1d

torch.nn.functional.**conv1d**(*input*, *weight*, *bias=None*, *stride=1*, *padding=0*, *dilation=1*, *groups=1*)

Applies a 1D convolution over an input signal composed of several input planes.

See *Conv1d* for details and output shape.

- **input** – input tensor of shape (minibatch x in_channels x iW)
- **weight** – filters of shape (out_channels, in_channels, kW)
- **bias** – optional bias of shape (out_channels)
- **stride** – the stride of the convolving kernel, default 1

Examples

```
>>> filters = autograd.Variable(torch.randn(33, 16, 3))
>>> inputs = autograd.Variable(torch.randn(20, 16, 50))
>>> F.conv1d(inputs, filters)
```

conv2d

torch.nn.functional.**conv2d**(*input*, *weight*, *bias=None*, *stride=1*, *padding=0*, *dilation=1*, *groups=1*)

Applies a 2D convolution over an input image composed of several input planes.

See *Conv2d* for details and output shape.

- **input** – input tensor (minibatch x in_channels x iH x iW)
- **weight** – filters tensor (out_channels, in_channels/groups, kH, kW)
- **bias** – optional bias tensor (out_channels)
- **stride** – the stride of the convolving kernel. Can be a single number or a tuple (sh x sw). Default: 1
- **padding** – implicit zero padding on the input. Can be a single number or a tuple. Default: 0
- **groups** – split input into groups, in_channels should be divisible by the number of groups

Examples

```
>>> # With square kernels and equal stride
>>> filters = autograd.Variable(torch.randn(8,4,3,3))
>>> inputs = autograd.Variable(torch.randn(1,4,5,5))
>>> F.conv2d(inputs, filters, padding=1)
```

conv3d

torch.nn.functional.**conv3d**(*input*, *weight*, *bias=None*, *stride=1*, *padding=0*, *dilation=1*, *groups=1*)

Applies a 3D convolution over an input image composed of several input planes.

See *Conv3d* for details and output shape.

- **input** – input tensor of shape (minibatch x in_channels x iT x iH x iW)
- **weight** – filters tensor of shape (out_channels, in_channels, kT, kH, kW)
- **bias** – optional bias tensor of shape (out_channels)
- **stride** – the stride of the convolving kernel. Can be a single number or a tuple (st x sh x sw). Default: 1
- **padding** – implicit zero padding on the input. Can be a single number or a tuple. Default: 0

Examples

```
>>> filters = autograd.Variable(torch.randn(33, 16, 3, 3, 3))
>>> inputs = autograd.Variable(torch.randn(20, 16, 50, 10, 20))
>>> F.conv3d(inputs, filters)
```

conv_transpose1d

torch.nn.functional.**conv_transpose1d**(*input*, *weight*, *bias=None*, *stride=1*, *padding=0*, *output_padding=0*, *groups=1*)

conv_transpose2d

torch.nn.functional.**conv_transpose2d**(*input*, *weight*, *bias=None*, *stride=1*, *padding=0*, *output_padding=0*, *groups=1*)

Applies a 2D transposed convolution operator over an input image composed of several input planes, sometimes also called "deconvolution".

See *ConvTranspose2d* for details and output shape.

- **input** – input tensor of shape (minibatch x in_channels x iH x iW)
- **weight** – filters of shape (in_channels x out_channels x kH x kW)
- **bias** – optional bias of shape (out_channels)
- **stride** – the stride of the convolving kernel, a single number or a tuple (sh x sw). Default: 1
- **padding** – implicit zero padding on the input, a single number or a tuple (padh x padw). Default: 0
- **groups** – split input into groups, in_channels should be divisible by the number of groups
- **output_padding** – A zero-padding of 0 <= padding < stride that should be added to the output. Can be a single number or a tuple. Default: 0

conv_transpose3d

torch.nn.functional.**conv_transpose3d**(*input*, *weight*, *bias=None*, *stride=1*, *padding=0*, *output_padding=0*, *groups=1*)

Applies a 3D transposed convolution operator over an input image composed of several input planes, sometimes also called "deconvolution"

See *ConvTranspose3d* for details and output shape.

- **input** – input tensor of shape (minibatch x in_channels x iT x iH x iW)
- **weight** – filters of shape (in_channels x out_channels x kH x kW)
- **bias** – optional bias of shape (out_channels)
- **stride** – the stride of the convolving kernel, a single number or a tuple (sh x sw). Default: 1
- **padding** – implicit zero padding on the input, a single number or a tuple (padh x padw). Default: 0

Pooling functions

avg_pool1d

torch.nn.functional.**avg_pool1d**(*input*, *kernel_size*, *stride=None*, *padding=0*, *ceil_mode=False*, *count_include_pad=True*)

Applies a 1D average pooling over an input signal composed of several input planes.

See *AvgPool1d* for details and output shape.

- **kernel_size** – the size of the window
- **stride** – the stride of the window. Default value is `kernel_size`
- **padding** – implicit zero padding to be added on both sides
- **ceil_mode** – when True, will use *ceil* instead of *floor* to compute the output shape
- **count_include_pad** – when True, will include the zero-padding in the averaging calculation

Example

```
>>> # pool of square window of size=3, stride=2
>>> input = Variable(torch.Tensor([[[1,2,3,4,5,6,7]]]))
>>> F.avg_pool1d(input, kernel_size=3, stride=2)
Variable containing:
(0 ,.,.) =
  2  4  6
[torch.FloatTensor of size 1x1x3]
```

avg_pool2d

`torch.nn.functional.`**avg_pool2d**(*input*, *kernel_size*, *stride=None*, *padding=0*, *ceil_mode=False*, *count_include_pad=True*)

Applies 2D average-pooling operation in kh x kw regions by step size dh x dw steps. The number of output features is equal to the number of input planes.

See *AvgPool2d* for details and output shape.

- **input** – input tensor (minibatch x in_channels x iH x iW)
- **kernel_size** – size of the pooling region, a single number or a tuple (kh x kw)
- **stride** – stride of the pooling operation, a single number or a tuple (sh x sw). Default is equal to kernel size
- **padding** – implicit zero padding on the input, a single number or a tuple (padh x padw), Default: 0
- **ceil_mode** – operation that defines spatial output shape
- **count_include_pad** – divide by the number of elements inside the original non-padded image or kh * kw

avg_pool3d

`torch.nn.functional.`**avg_pool3d**(*input*, *kernel_size*, *stride=None*)

Applies 3D average-pooling operation in kt x kh x kw regions by step size kt x dh x dw steps. The number of output features is equal to the number of input planes / dt.

max_pool1d

torch.nn.functional.**max_pool1d**(*input*, *kernel_size*, *stride=None*, *padding=0*, *dilation=1*, *ceil_mode=False*, *return_indices=False*)

max_pool2d

torch.nn.functional.**max_pool2d**(*input*, *kernel_size*, *stride=None*, *padding=0*, *dilation=1*, *ceil_mode=False*, *return_indices=False*)

max_pool3d

torch.nn.functional.**max_pool3d**(*input*, *kernel_size*, *stride=None*, *padding=0*, *dilation=1*, *ceil_mode=False*, *return_indices=False*)

max_unpool1d

torch.nn.functional.**max_unpool1d**(*input*, *indices*, *kernel_size*, *stride=None*, *padding=0*, *output_size=None*)

max_unpool2d

torch.nn.functional.**max_unpool2d**(*input*, *indices*, *kernel_size*, *stride=None*, *padding=0*, *output_size=None*)

max_unpool3d

torch.nn.functional.**max_unpool3d**(*input*, *indices*, *kernel_size*, *stride=None*, *padding=0*, *output_size=None*)

lp_pool2d

torch.nn.functional.**lp_pool2d**(*input*, *norm_type*, *kernel_size*, *stride=None*, *ceil_mode=False*)

adaptive_max_pool1d

torch.nn.functional.**adaptive_max_pool1d**(*input*, *output_size*, *return_indices=False*)

Applies a 1D adaptive max pooling over an input signal composed of several input planes.

See *AdaptiveMaxPool1d* for details and output shape.

- **output_size** – the target output size (single integer)
- **return_indices** – whether to return pooling indices

adaptive_max_pool2d

torch.nn.functional.**adaptive_max_pool2d**(*input*, *output_size*, *return_indices=False*)

Applies a 2D adaptive max pooling over an input signal composed of several input planes.

See *AdaptiveMaxPool2d* for details and output shape.

- **output_size** – the target output size (single integer or double-integer tuple)
- **return_indices** – whether to return pooling indices

adaptive_avg_pool1d

torch.nn.functional.**adaptive_avg_pool1d**(*input*, *output_size*)

Applies a 1D adaptive average pooling over an input signal composed of several input planes.

See *AdaptiveAvgPool1d* for details and output shape.

output_size – the target output size (single integer)

adaptive_avg_pool2d

torch.nn.functional.**adaptive_avg_pool2d**(*input*, *output_size*)

Applies a 2D adaptive average pooling over an input signal composed of several input planes.

See *AdaptiveAvgPool2d* for details and output shape.

output_size – the target output size (single integer or double-integer tuple)

Non-linear activation functions

threshold

torch.nn.functional.**threshold**(*input*, *threshold*, *value*, *inplace=False*)

relu

torch.nn.functional.**relu**(*input*, *inplace=False*)

hardtanh

torch.nn.functional.**hardtanh**(*input*, *min_val=-1.0*, *max_val=1.0*, *inplace=False*)

relu6

torch.nn.functional.**relu6**(*input*, *inplace=False*)

elu

torch.nn.functional.**elu**(*input*, *alpha=1.0*, *inplace=False*)

leaky_relu

torch.nn.functional.**leaky_relu**(*input*, *negative_slope=0.01*, *inplace=False*)

prelu

torch.nn.functional.**prelu**(*input*, *weight*)

rrelu

torch.nn.functional.**rrelu**(*input*, *lower=0.125*, *upper=0.3333333333333333*, *training=False*, *inplace=False*)

logsigmoid

torch.nn.functional.**logsigmoid**(*input*)

hardshrink

torch.nn.functional.**hardshrink**(*input*, *lambd=0.5*)

tanhshrink

torch.nn.functional.**tanhshrink**(*input*)

softsign

torch.nn.functional.**softsign**(*input*)

softplus

torch.nn.functional.**softplus**(*input*, *beta=1*, *threshold=20*)

softmin

torch.nn.functional.**softmin**(*input*)

softmax

torch.nn.functional.**softmax**(*input*)

softshrink

torch.nn.functional.**softshrink**(*input*, *lambd=0.5*)

log_softmax

torch.nn.functional.**log_softmax**(*input*)

tanh

torch.nn.functional.**tanh**(*input*)

sigmoid

torch.nn.functional.**sigmoid**(*input*)

Normalization functions

batch_norm

torch.nn.functional.**batch_norm**(*input*, *running_mean*, *running_var*, *weight=None*, *bias=None*, *training=False*, *momentum=0.1*, *eps=1e-05*)

Linear functions

linear

torch.nn.functional.**linear**(*input*, *weight*, *bias=None*)

Dropout functions

dropout

torch.nn.functional.**dropout**(*input*, *p=0.5*, *training=False*, *inplace=False*)

Distance functions

pairwise_distance

torch.nn.functional.**pairwise_distance**(*x1*, *x2*, *p=2*, *eps=1e-06*)

Computes the batchwise pairwise distance between vectors v1,v2:

$$\|x\|_p := \left(\sum_{i=1}^{n} |x_i|^p \right)^{1/p}$$

Args: x (Tensor): input tensor containing the two input batches p (real): the norm degree. Default: 2

Shape:

- Input: (N, D) where *D = vector dimension*
- Output: $(N, 1)$

```
>>> input1 = autograd.Variable(torch.randn(100, 128))
>>> input2 = autograd.Variable(torch.randn(100, 128))
>>> output = F.pairwise_distance(input1, input2, p=2)
>>> output.backward()
```

Loss functions

nll_loss

torch.nn.functional.**nll_loss**(*input*, *target*, *weight=None*, *size_average=True*)

The negative log likelihood loss.

See *NLLLoss* for details.

- **input** – (N, C) where *C = number of classes*
- **target** – (N) where each value is *0 <= targets[i] <= C-1*
- **weight** (*Variable, optional*) – a manual rescaling weight given to each class. If given, has to be a Variable of size "nclasses"
- **size_average** (*bool, optional*) – 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.

weight – the class-weights given as input to the constructor

Example

```
>>> # input is of size nBatch x nClasses = 3 x 5
>>> input = autograd.Variable(torch.randn(3, 5))
>>> # each element in target has to have 0 <= value < nclasses
>>> target = autograd.Variable(torch.LongTensor([1, 0, 4]))
>>> output = F.nll_loss(F.log_softmax(input), target)
>>> output.backward()
```

kl_div

torch.nn.functional.**kl_div**(*input*, *target*, *size_average=True*)

The Kullback-Leibler divergence Loss.

See *KLDivLoss* for details.

- **input** – Variable of arbitrary shape
- **target** – Variable of the same shape as input
- **size_average** – if True the output is divided by the number of elements in input tensor

cross_entropy

torch.nn.functional.**cross_entropy**(*input*, *target*, *weight=None*, *size_average=True*)

This criterion combines *log_softmax* and *nll_loss* in one single class.

See *torch.nn.CrossEntropyLoss* for details.

- **input** – Variable (N, C) where *C = number of classes*
- **target** – Variable (N) where each value is *0 <= targets[i] <= C-1*
- **weight** (*Tensor, optional*) – a manual rescaling weight given to each class. If given, has to be a Tensor of size "nclasses"
- **size_average** (*bool, optional*) – 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.

binary_cross_entropy

torch.nn.functional.**binary_cross_entropy**(*input*, *target*, *weight=None*, *size_average=True*)

Function that measures the Binary Cross Entropy between the target and the output:

See *BCELoss* for details.

- **input** – Variable of arbitrary shape
- **target** – Variable of the same shape as input
- **weight** (*Variable, optional*) – a manual rescaling weight if provided it's repeated to match input tensor shape
- **size_average** (*bool, optional*) – 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.

smooth_l1_loss

torch.nn.functional.**smooth_l1_loss**(*input*, *target*, *size_average=True*)

Vision functions

pixel_shuffle

torch.nn.functional.**pixel_shuffle**(*input*, *upscale_factor*)

Rearranges elements in a tensor of shape `[*, C*r^2, H, W]` to a tensor of shape `[C, H*r, W*r]`.

See *PixelShuffle* for details.

- **input** (Variable) – Input
- **upscale_factor** (*int*) – factor to increase spatial resolution by

Examples

```
>>> ps = nn.PixelShuffle(3)
>>> input = autograd.Variable(torch.Tensor(1, 9, 4, 4))
>>> output = ps(input)
>>> print(output.size())
torch.Size([1, 1, 12, 12])
```

pad

torch.nn.functional.**pad**(*input*, *pad*, *mode='constant'*, *value=0*)

Pads tensor.

Currently only 2D and 3D padding supported. In case of 4D input tensor pad should be in form (pad_l, pad_r, pad_t, pad_b) In case of 5D pad should be (pleft, pright, ptop, pbottom, pfront, pback)

- **input** (Variable) – 4D or 5D tensor
- **pad** (*tuple*) – 4-elem or 6-elem tuple
- **mode** – 'constant', 'reflect' or 'replicate'
- **value** – fill value for 'constant' padding

CHAPTER 11

torch.nn.init

torch.nn.init.**uniform**(*tensor*, *a=0*, *b=1*)
Fills the input Tensor or Variable with values drawn from a uniform U(a,b)

- **tensor** – a n-dimension torch.Tensor
- **a** – the lower bound of the uniform distribution
- **b** – the upper bound of the uniform distribution

Examples

```
>>> w = torch.Tensor(3, 5)
>>> nn.init.uniform(w)
```

torch.nn.init.**normal**(*tensor*, *mean=0*, *std=1*)
Fills the input Tensor or Variable with values drawn from a normal distribution with the given mean and std

- **tensor** – a n-dimension torch.Tensor
- **mean** – the mean of the normal distribution
- **std** – the standard deviation of the normal distribution

Examples

```
>>> w = torch.Tensor(3, 5)
>>> nn.init.normal(w)
```

torch.nn.init.**constant**(*tensor*, *val*)
Fills the input Tensor or Variable with the value *val*

- **tensor** – a n-dimension torch.Tensor
- **val** – the value to fill the tensor with

Examples

```
>>> w = torch.Tensor(3, 5)
>>> nn.init.constant(w)
```

torch.nn.init.**xavier_uniform**(*tensor*, *gain=1*)

Fills the input Tensor or Variable with values according to the method described in "Understanding the difficulty of training deep feedforward neural networks" - Glorot, X. and Bengio, Y., using a uniform distribution. The resulting tensor will have values sampled from U(-a, a) where a = gain * sqrt(2/(fan_in + fan_out)) * sqrt(3)

- **tensor** – a n-dimension torch.Tensor
- **gain** – an optional scaling factor to be applied

Examples

```
>>> w = torch.Tensor(3, 5)
>>> nn.init.xavier_uniform(w, gain=math.sqrt(2.0))
```

torch.nn.init.**xavier_normal**(*tensor*, *gain=1*)

Fills the input Tensor or Variable with values according to the method described in "Understanding the difficulty of training deep feedforward neural networks" - Glorot, X. and Bengio, Y., using a normal distribution. The resulting tensor will have values sampled from normal distribution with mean=0 and std = gain * sqrt(2/(fan_in + fan_out))

- **tensor** – a n-dimension torch.Tensor
- **gain** – an optional scaling factor to be applied

Examples

```
>>> w = torch.Tensor(3, 5)
>>> nn.init.xavier_normal(w)
```

torch.nn.init.**kaiming_uniform**(*tensor*, *a=0*, *mode='fan_in'*)

Fills the input Tensor or Variable with values according to the method described in "Delving deep into rectifiers: Surpassing human-level performance on ImageNet classification" - He, K. et al using a uniform distribution. The resulting tensor will have values sampled from U(-bound, bound) where bound = sqrt(2/((1 + a^2) * fan_in)) * sqrt(3)

- **tensor** – a n-dimension torch.Tensor
- **a** – the coefficient of the slope of the rectifier used after this layer (0 for ReLU by default)

- **mode** – either 'fan_in' (default) or 'fan_out'. Choosing *fan_in* preserves the magnitude of the variance of the weights in the forward pass. Choosing *fan_out* preserves the magnitudes in the backwards pass.

Examples

```
>>> w = torch.Tensor(3, 5)
>>> nn.init.kaiming_uniform(w, mode='fan_in')
```

torch.nn.init.**kaiming_normal**(*tensor*, *a=0*, *mode='fan_in'*)

Fills the input Tensor or Variable with values according to the method described in "Delving deep into rectifiers: Surpassing human-level performance on ImageNet classification" - He, K. et al using a normal distribution. The resulting tensor will have values sampled from normal distribution with mean=0 and std = sqrt(2/((1 + a^2) * fan_in))

- **tensor** – a n-dimension torch.Tensor
- **a** – the coefficient of the slope of the rectifier used after this layer (0 for ReLU by default)
- **mode** – either 'fan_in' (default) or 'fan_out'. Choosing *fan_in* preserves the magnitude of the variance of the weights in the forward pass. Choosing *fan_out* preserves the magnitudes in the backwards pass.

Examples

```
>>> w = torch.Tensor(3, 5)
>>> nn.init.kaiming_normal(w, mode='fan_out')
```

torch.nn.init.**orthogonal**(*tensor*, *gain=1*)

Fills the input Tensor or Variable with a (semi) orthogonal matrix. The input tensor must have at least 2 dimensions, and for tensors with more than 2 dimensions the trailing dimensions are flattened. viewed as 2D representation with rows equal to the first dimension and columns equal to the product of as a sparse matrix, where the non-zero elements will be drawn from a normal distribution with mean=0 and std='std'. Reference: "Exact solutions to the nonlinear dynamics of learning in deep linear neural networks"-Saxe, A. et al.

- **tensor** – a n-dimension torch.Tensor, where n >= 2
- **gain** – optional gain to be applied

Examples

```
>>> w = torch.Tensor(3, 5)
>>> nn.init.orthogonal(w)
```

torch.nn.init.**sparse**(*tensor*, *sparsity*, *std=0.01*)

Fills the 2D input Tensor or Variable as a sparse matrix, where the non-zero elements will be drawn from a normal distribution with mean=0 and std='std'.

- **tensor** – a n-dimension torch.Tensor

- **sparsity** – The fraction of elements in each column to be set to zero
- **std** – the standard deviation of the normal distribution used to generate the non-zero values

Examples

```
>>> w = torch.Tensor(3, 5)
>>> nn.init.sparse(w, sparsity=0.1)
```

CHAPTER 12

torch.optim

torch.optim is a package implementing various optimization algorithms. Most commonly used methods are already supported, and the interface is general enough, so that more sophisticated ones can be also easily integrated in the future.

How to use an optimizer

To use *torch.optim* you have to construct an optimizer object, that will hold the current state and will update the parameters based on the computed gradients.

Constructing it

To construct an *Optimizer* you have to give it an iterable containing the parameters (all should be *Variable* s) to optimize. Then, you can specify optimizer-specific options such as the learning rate, weight decay, etc.

Example:

```
optimizer = optim.SGD(model.parameters(), lr = 0.01, momentum=0.9)
optimizer = optim.Adam([var1, var2], lr = 0.0001)
```

Per-parameter options

Optimizer s also support specifying per-parameter options. To do this, instead of passing an iterable of *Variable* s, pass in an iterable of `dict` s. Each of them will define a separate parameter group, and should contain a `params` key, containing a list of parameters belonging to it. Other keys should match the keyword arguments accepted by the optimizers, and will be used as optimization options for this group.

: You can still pass options as keyword arguments. They will be used as defaults, in the groups that didn't override them. This is useful when you only want to vary a single option, while keeping all others consistent between parameter

groups.

For example, this is very useful when one wants to specify per-layer learning rates:

```
optim.SGD([
                {'params': model.base.parameters()},
                {'params': model.classifier.parameters(), 'lr': 1e-3}
            ], lr=1e-2, momentum=0.9)
```

This means that `model.base`'s parameters will use the default learning rate of `1e-2`, `model.classifier`'s parameters will use a learning rate of `1e-3`, and a momentum of `0.9` will be used for all parameters

Taking an optimization step

All optimizers implement a *`step()`* method, that updates the parameters. It can be used in two ways:

optimizer.step()

This is a simplified version supported by most optimizers. The function can be called once the gradients are computed using e.g. *`backward()`*.

Example:

```
for input, target in dataset:
    optimizer.zero_grad()
    output = model(input)
    loss = loss_fn(output, target)
    loss.backward()
    optimizer.step()
```

optimizer.step(closure)

Some optimization algorithms such as Conjugate Gradient and LBFGS need to reevaluate the function multiple times, so you have to pass in a closure that allows them to recompute your model. The closure should clear the gradients, compute the loss, and return it.

Example:

```
for input, target in dataset:
    def closure():
        optimizer.zero_grad()
        output = model(input)
        loss = loss_fn(output, target)
        loss.backward()
        return loss
    optimizer.step(closure)
```

Algorithms

class `torch.optim.`**`Optimizer`**(*params*, *defaults*)
Base class for all optimizers.

- **params** (*iterable*) – an iterable of `Variable` s or `dict` s. Specifies what Variables should be optimized.
- **defaults** – (dict): a dict containing default values of optimization options (used when a parameter group doesn't specify them).

load_state_dict(*state_dict*)
Loads the optimizer state.

state_dict (*dict*) – optimizer state. Should be an object returned from a call to *state_dict()*.

state_dict()
Returns the state of the optimizer as a `dict`.

It contains two entries:

•**state - a dict holding current optimization state. Its content** differs between optimizer classes.

•param_groups - a dict containig all parameter groups

step(*closure*)
Performs a single optimization step (parameter update).

closure (*callable*) – A closure that reevaluates the model and returns the loss. Optional for most optimizers.

zero_grad()
Clears the gradients of all optimized `Variable` s.

class `torch.optim.`**Adadelta**(*params*, *lr=1.0*, *rho=0.9*, *eps=1e-06*, *weight_decay=0*)
Implements Adadelta algorithm.

It has been proposed in ADADELTA: An Adaptive Learning Rate Method.

- **params** (*iterable*) – iterable of parameters to optimize or dicts defining parameter groups
- **rho** (*float, optional*) – coefficient used for computing a running average of squared gradients (default: 0.9)
- **eps** (*float, optional*) – term added to the denominator to improve numerical stability (default: 1e-6)
- **lr** (*float, optional*) – coefficient that scale delta before it is applied to the parameters (default: 1.0)
- **weight_decay** (*float, optional*) – weight decay (L2 penalty) (default: 0)

step(*closure=None*)
Performs a single optimization step.

closure (*callable, optional*) – A closure that reevaluates the model and returns the loss.

class `torch.optim.`**Adagrad**(*params*, *lr=0.01*, *lr_decay=0*, *weight_decay=0*)
Implements Adagrad algorithm.

It has been proposed in Adaptive Subgradient Methods for Online Learning and Stochastic Optimization.

- **params** (*iterable*) – iterable of parameters to optimize or dicts defining parameter groups
- **lr** (*float, optional*) – learning rate (default: 1e-2)
- **lr_decay** (*float, optional*) – learning rate decay (default: 0)
- **weight_decay** (*float, optional*) – weight decay (L2 penalty) (default: 0)

step(*closure=None*)
Performs a single optimization step.

closure (*callable, optional*) – A closure that reevaluates the model and returns the loss.

class torch.optim.**Adam**(*params*, *lr=0.001*, *betas=(0.9, 0.999)*, *eps=1e-08*, *weight_decay=0*)
Implements Adam algorithm.

It has been proposed in Adam: A Method for Stochastic Optimization.

- **params** (*iterable*) – iterable of parameters to optimize or dicts defining parameter groups
- **lr** (*float, optional*) – learning rate (default: 1e-3)
- **betas** (*Tuple[float, float], optional*) – coefficients used for computing running averages of gradient and its square (default: (0.9, 0.999))
- **eps** (*float, optional*) – term added to the denominator to improve numerical stability (default: 1e-8)
- **weight_decay** (*float, optional*) – weight decay (L2 penalty) (default: 0)

step(*closure=None*)
Performs a single optimization step.

closure (*callable, optional*) – A closure that reevaluates the model and returns the loss.

class torch.optim.**Adamax**(*params*, *lr=0.002*, *betas=(0.9, 0.999)*, *eps=1e-08*, *weight_decay=0*)
Implements Adamax algorithm (a variant of Adam based on infinity norm).

It has been proposed in Adam: A Method for Stochastic Optimization.

- **params** (*iterable*) – iterable of parameters to optimize or dicts defining parameter groups
- **lr** (*float, optional*) – learning rate (default: 2e-3)
- **betas** (*Tuple[float, float], optional*) – coefficients used for computing running averages of gradient and its square
- **eps** (*float, optional*) – term added to the denominator to improve numerical stability (default: 1e-8)
- **weight_decay** (*float, optional*) – weight decay (L2 penalty) (default: 0)

step(*closure=None*)
Performs a single optimization step.

closure (*callable, optional*) – A closure that reevaluates the model and returns the loss.

class torch.optim.**ASGD**(*params*, *lr=0.01*, *lambd=0.0001*, *alpha=0.75*, *t0=1000000.0*, *weight_decay=0*)
Implements Averaged Stochastic Gradient Descent.

It has been proposed in Acceleration of stochastic approximation by averaging.

- **params** (*iterable*) – iterable of parameters to optimize or dicts defining parameter groups
- **lr** (*float, optional*) – learning rate (default: 1e-2)
- **lambd** (*float, optional*) – decay term (default: 1e-4)
- **alpha** (*float, optional*) – power for eta update (default: 0.75)
- **t0** (*float, optional*) – point at which to start averaging (default: 1e6)
- **weight_decay** (*float, optional*) – weight decay (L2 penalty) (default: 0)

step(*closure=None*)
Performs a single optimization step.

closure (*callable, optional*) – A closure that reevaluates the model and returns the loss.

class torch.optim.**LBFGS**(*params*, *lr=1*, *max_iter=20*, *max_eval=None*, *tolerance_grad=1e-05*, *tolerance_change=1e-09*, *history_size=100*, *line_search_fn=None*)
Implements L-BFGS algorithm.

: This optimizer doesn't support per-parameter options and parameter groups (there can be only one).

: Right now all parameters have to be on a single device. This will be improved in the future.

: This is a very memory intensive optimizer (it requires additional `param_bytes * (history_size + 1)` bytes). If it doesn't fit in memory try reducing the history size, or use a different algorithm.

- **lr** (*float*) – learning rate (default: 1)
- **max_iter** (*int*) – maximal number of iterations per optimization step (default: 20)
- **max_eval** (*int*) – maximal number of function evaluations per optimization step (default: max_iter * 1.25).
- **tolerance_grad** (*float*) – termination tolerance on first order optimality (default: 1e-5).
- **tolerance_change** (*float*) – termination tolerance on function value/parameter changes (default: 1e-9).
- **history_size** (*int*) – update history size (default: 100).

step(*closure*)
Performs a single optimization step.

closure (*callable*) – A closure that reevaluates the model and returns the loss.

class torch.optim.**RMSprop**(*params*, *lr=0.01*, *alpha=0.99*, *eps=1e-08*, *weight_decay=0*, *momentum=0*, *centered=False*)

Implements RMSprop algorithm.

Proposed by G. Hinton in his course.

The centered version first appears in Generating Sequences With Recurrent Neural Networks.

- **params** (*iterable*) – iterable of parameters to optimize or dicts defining parameter groups
- **lr** (*float, optional*) – learning rate (default: 1e-2)
- **momentum** (*float, optional*) – momentum factor (default: 0)
- **alpha** (*float, optional*) – smoothing constant (default: 0.99)
- **eps** (*float, optional*) – term added to the denominator to improve numerical stability (default: 1e-8)
- **centered** (*bool, optional*) – if True, compute the centered RMSProp, the gradient is normalized by an estimation of its variance
- **weight_decay** (*float, optional*) – weight decay (L2 penalty) (default: 0)

step(*closure=None*)

Performs a single optimization step.

closure (*callable, optional*) – A closure that reevaluates the model and returns the loss.

class torch.optim.**Rprop**(*params*, *lr=0.01*, *etas=(0.5, 1.2)*, *step_sizes=(1e-06, 50)*)

Implements the resilient backpropagation algorithm.

- **params** (*iterable*) – iterable of parameters to optimize or dicts defining parameter groups
- **lr** (*float, optional*) – learning rate (default: 1e-2)
- **etas** (*Tuple[float, float], optional*) – pair of (etaminus, etaplis), that are multiplicative increase and decrease factors (default: (0.5, 1.2))
- **step_sizes** (*Tuple[float, float], optional*) – a pair of minimal and maximal allowed step sizes (default: (1e-6, 50))

step(*closure=None*)

Performs a single optimization step.

closure (*callable, optional*) – A closure that reevaluates the model and returns the loss.

class torch.optim.**SGD**(*params*, *lr=<object object>*, *momentum=0*, *dampening=0*, *weight_decay=0*, *nesterov=False*)

Implements stochastic gradient descent (optionally with momentum).

Nesterov momentum is based on the formula from On the importance of initialization and momentum in deep learning.

- **params** (*iterable*) – iterable of parameters to optimize or dicts defining parameter groups

- **lr** (*float*) – learning rate
- **momentum** (*float, optional*) – momentum factor (default: 0)
- **weight_decay** (*float, optional*) – weight decay (L2 penalty) (default: 0)
- **dampening** (*float, optional*) – dampening for momentum (default: 0)
- **nesterov** (*bool, optional*) – enables Nesterov momentum (default: False)

Example

```
>>> optimizer = torch.optim.SGD(model.parameters(), lr=0.1, momentum=0.9)
>>> optimizer.zero_grad()
>>> loss_fn(model(input), target).backward()
>>> optimizer.step()
```

step(*closure=None*)

Performs a single optimization step.

closure (*callable, optional*) – A closure that reevaluates the model and returns the loss.

CHAPTER 13

Automatic differentiation package - torch.autograd

torch.autograd provides classes and functions implementing automatic differentiation of arbitrary scalar valued functions. It requires minimal changes to the existing code - you only need to wrap all tensors in *Variable* objects.

torch.autograd.**backward**(*variables*, *grad_variables*, *retain_variables=False*)

Computes the sum of gradients of given variables w.r.t. graph leaves.

The graph is differentiated using the chain rule. If any of `variables` are non-scalar (i.e. their data has more than one element) and require gradient, the function additionaly requires specifying `grad_variables`. It should be a sequence of matching length, that containins gradient of the differentiated function w.r.t. corresponding variables (`None` is an acceptable value for all variables that don't need gradient tensors).

This function accumulates gradients in the leaves - you might need to zero them before calling it.

- **variables** (*sequence of Variable*) – Variables of which the derivative will be computed.
- **grad_variables** (*sequence of Tensor*) – Gradients w.r.t. each element of corresponding variables. Required only for non-scalar variables that require gradient.
- **retain_variables** (*bool*) – If `True`, buffers necessary for computing gradients won't be freed after use. It is only necessary to specify `True` if you want to differentiate some subgraph multiple times.

Variable

API compatibility

Variable API is nearly the same as regular Tensor API (with the exception of a couple in-place methods, that would overwrite inputs required for gradient computation). In most cases Tensors can be safely replaced with Variables and the code will remain to work just fine. Because of this, we're not documenting all the operations on variables, and you should refere to *torch.Tensor* docs for this purpose.

In-place operations on Variables

Supporting in-place operations in autograd is a hard matter, and we discourage their use in most cases. Autograd's aggressive buffer freeing and reuse makes it very efficient and there are very few occasions when in-place operations actually lower memory usage by any significant amount. Unless you're operating under heavy memory pressure, you might never need to use them.

In-place correctness checks

All *Variable* s keep track of in-place operations applied to them, and if the implementation detects that a variable was saved for backward in one of the functions, but it was modified in-place afterwards, an error will be raised once backward pass is started. This ensures that if you're using in-place functions and not seing any errors, you can be sure that the computed gradients are correct.

class `torch.autograd.`**Variable**

Wraps a tensor and records the operations applied to it.

Variable is a thin wrapper around a Tensor object, that also holds the gradient w.r.t. to it, and a reference to a function that created it. This reference allows retracing the whole chain of operations that created the data. If the Variable has been created by the user, its creator will be `None` and we call such objects *leaf* Variables.

Since autograd only supports scalar valued function differentiation, grad size always matches the data size. Also, grad is normally only allocated for leaf variables, and will be always zero otherwise.

- **data** – Wrapped tensor of any type.
- **grad** – Variable holding the gradient of type and location matching the `.data`. This attribute is lazily allocated and can't be reassigned.
- **requires_grad** – Boolean indicating whether the Variable has been created by a subgraph containing any Variable, that requires it. See *Excluding subgraphs from backward* for more details. Can be changed only on leaf Variables.
- **volatile** – Boolean indicating that the Variable should be used in inference mode, i.e. don't save the history. See *Excluding subgraphs from backward* for more details. Can be changed only on leaf Variables.
- **creator** – Function of which the variable was an output. For leaf (user created) variables it's `None`. Read-only attribute.

- **data** (*any tensor class*) – Tensor to wrap.
- **requires_grad** (*bool*) – Value of the requires_grad flag. **Keyword only.**
- **volatile** (*bool*) – Value of the volatile flag. **Keyword only.**

backward(*gradient=None*, *retain_variables=False*)

Computes the gradient of current variable w.r.t. graph leaves.

The graph is differentiated using the chain rule. If the variable is non-scalar (i.e. its data has more than one element) and requires gradient, the function additionaly requires specifying `gradient`. It should be a tensor of matching type and location, that contains the gradient of the differentiated function w.r.t. `self`.

This function accumulates gradients in the leaves - you might need to zero them before calling it.

- **gradient** (Tensor) – Gradient of the differentiated function w.r.t. the data. Required only if the data has more than one element. Type and location should match these of `self.data`.
- **retain_variables** (*bool*) – If `True`, buffers necessary for computing gradients won't be freed after use. It is only necessary to specify `True` if you want to differentiate some subgraph multiple times (in some cases it will be much more efficient to use *autograd.backward*).

detach()

Returns a new Variable, detached from the current graph.

Result will never require gradient. If the input is volatile, the output will be volatile too.

: Returned Variable uses the same data tensor, as the original one, and in-place modifications on either of them will be seen, and may trigger errors in correctness checks.

---

detach_()

Detaches the Variable from the graph that created it, making it a leaf.

register_hook(*hook*)

Registers a backward hook.

The hook will be called every time a gradient with respect to the variable is computed. The hook should have the following signature:

```
hook(grad) -> Variable or None
```

The hook should not modify its argument, but it can optionally return a new gradient which will be used in place of `grad`.

This function returns a handle with a method `handle.remove()` that removes the hook from the module.

Example

```
>>> v = Variable(torch.Tensor([0, 0, 0]), requires_grad=True)
>>> h = v.register_hook(lambda grad: grad * 2)  # double the gradient
>>> v.backward(torch.Tensor([1, 1, 1]))
>>> v.grad.data
 2
 2
 2
[torch.FloatTensor of size 3]
>>> h.remove()  # removes the hook
```

reinforce(*reward*)

Registers a reward obtained as a result of a stochastic process.

Differentiating stochastic nodes requires providing them with reward value. If your graph contains any stochastic operations, you should call this function on their outputs. Otherwise an error will be raised.

reward (Tensor) – Tensor with per-element rewards. It has to match the device location and shape of Variable's data.

Function

class `torch.autograd.`**Function**

Records operation history and defines formulas for differentiating ops.

Every operation performed on *Variable* s creates a new function object, that performs the computation, and records that it happened. The history is retained in the form of a DAG of functions, with edges denoting data dependencies (`input <- output`). Then, when backward is called, the graph is processed in the topological ordering, by calling *backward()* methods of each *Function* object, and passing returned gradients on to next *Function* s.

Normally, the only way users interact with functions is by creating subclasses and defining new operations. This is a recommended way of extending torch.autograd.

Since Function logic is a hotspot in most scripts, almost all of it was moved to our C backend, to ensure that the framework overhead is minimal.

Each function is meant to be used only once (in the forward pass).

- **saved_tensors** – Tuple of Tensors that were saved in the call to *forward()*.
- **needs_input_grad** – Tuple of booleans of length `num_inputs`, indicating whether a given input requires gradient. This can be used to optimize buffers saved for backward, and ignoring gradient computation in *backward()*.
- **num_inputs** – Number of inputs given to *forward()*.
- **num_outputs** – Number of tensors returned by *forward()*.
- **requires_grad** – Boolean indicating whether the *backward()* will ever need to be called.
- **previous_functions** – Tuple of (int, Function) pairs of length `num_inputs`. Each entry contains a reference to a *Function* that created corresponding input, and an index of the previous function output that's been used.

backward(**grad_output*)

Defines a formula for differentiating the operation.

This function is to be overriden by all subclasses.

All arguments are tensors. It has to accept exactly as many arguments, as many outputs did *forward()* return, and it should return as many tensors, as there were inputs to *forward()*. Each argument is the gradient w.r.t the given output, and each returned value should be the gradient w.r.t. the corresponding input.

forward(**input*)

Performs the operation.

This function is to be overriden by all subclasses.

It can take and return an arbitrary number of tensors.

mark_dirty(**args*)

Marks given tensors as modified in an in-place operation.

This should be called at most once, only from inside the *forward()* method, and all arguments should be inputs.

Every tensor that's been modified in-place in a call to *forward()* should be given to this function, to ensure correcness of our checks. It doesn't matter wheter the function is called before or after modification.

mark_non_differentiable(**args*)
Marks outputs as non-differentiable.

This should be called at most once, only from inside the *forward()* **method, and all arguments should be outputs.**

This will mark outputs as not requiring gradients, increasing the efficiency of backward computation. You still need to accept a gradient for each output in *backward()*, but it's always going to be `None`.

This is used e.g. for indices returned from a max *Function*.

mark_shared_storage(**pairs*)
Marks that given pairs of distinct tensors are sharing storage.

This should be called at most once, only from inside the *forward()* **method, and all arguments should be pairs of (input, output).**

If some of the outputs are going to be tensors sharing storage with some of the inputs, all pairs of (input_arg, output_arg) should be given to this function, to ensure correctness checking of in-place modification. The only exception is when an output is exactly the same tensor as input (e.g. in-place ops). In such case it's easy to conclude that they're sharing data, so we don't require specifying such dependencies.

This function is not needed in most functions. It's primarily used in indexing and transpose ops.

save_for_backward(**tensors*)
Saves given tensors for a future call to *backward()*.

This should be called at most once, and only from inside the *forward()* **method.**

Later, saved tensors can be accessed through the `saved_tensors` attribute. Before returning them to the user, a check is made, to ensure they weren't used in any in-place operation that modified their content.

Arguments can also be `None`.

CHAPTER 14

Multiprocessing package - torch.multiprocessing

torch.multiprocessing is a wrapper around the native `multiprocessing` module. It registers custom reducers, that use shared memory to provide shared views on the same data in different processes. Once the tensor/storage is moved to shared_memory (see *share_memory_()*), it will be possible to send it to other processes without making any copies.

The API is 100% compatible with the original module - it's enough to change `import multiprocessing` to `import torch.multiprocessing` to have all the tensors sent through the queues or shared via other mechanisms, moved to shared memory.

Because of the similarity of APIs we do not document most of this package contents, and we recommend referring to very good docs of the original module.

: If the main process exits abruptly (e.g. because of an incoming signal), Python's `multiprocessing` sometimes fails to clean up its children. It's a known caveat, so if you're seeing any resource leaks after interrupting the interpreter, it probably means that this has just happened to you.

Strategy management

`torch.multiprocessing.`**`get_all_sharing_strategies`**`()`
Returns a set of sharing strategies supported on a current system.

`torch.multiprocessing.`**`get_sharing_strategy`**`()`
Returns the current strategy for sharing CPU tensors.

`torch.multiprocessing.`**`set_sharing_strategy`**`(`*new_strategy*`)`
Sets the strategy for sharing CPU tensors.

new_strategy (*str*) – Name of the selected strategy. Should be one of the values returned by *get_all_sharing_strategies()*.

Sharing CUDA tensors

Sharing CUDA tensors between processes is supported only in Python 3, using a `spawn` or `forkserver` start methods. `multiprocessing` in Python 2 can only create subprocesses using `fork`, and it's not supported by the CUDA runtime.

: CUDA API requires that the allocation exported to other processes remains valid as long as it's used by them. You should be careful and ensure that CUDA tensors you shared don't go out of scope as long as it's necessary. This shouldn't be a problem for sharing model parameters, but passing other kinds of data should be done with care. Note that this restriction doesn't apply to shared CPU memory.

Sharing strategies

This section provides a brief overview into how different sharing strategies work. Note that it applies only to CPU tensor - CUDA tensors will always use the CUDA API, as that's the only way they can be shared.

File descriptor - `file_descriptor`

: This is the default strategy (except for macOS and OS X where it's not supported).

This strategy will use file descriptors as shared memory handles. Whenever a storage is moved to shared memory, a file descriptor obtained from `shm_open` is cached with the object, and when it's going to be sent to other processes, the file descriptor will be transferred (e.g. via UNIX sockets) to it. The receiver will also cache the file descriptor and `mmap` it, to obtain a shared view onto the storage data.

Note that if there will be a lot of tensors shared, this strategy will keep a large number of file descriptors open most of the time. If your system has low limits for the number of open file descriptors, and you can't rise them, you should use the `file_system` strategy.

File system - `file_system`

This strategy will use file names given to `shm_open` to identify the shared memory regions. This has a benefit of not requiring the implementation to cache the file descriptors obtained from it, but at the same time is prone to shared memory leaks. The file can't be deleted right after its creation, because other processes need to access it to open their views. If the processes fatally crash, or are killed, and don't call the storage destructors, the files will remain in the system. This is very serious, because they keep using up the memory until the system is restarted, or they're freed manually.

To counter the problem of shared memory file leaks, *`torch.multiprocessing`* will spawn a daemon named `torch_shm_manager` that will isolate itself from the current process group, and will keep track of all shared memory allocations. Once all processes connected to it exit, it will wait a moment to ensure there will be no new connections, and will iterate over all shared memory files allocated by the group. If it finds that any of them still exist, they will be deallocated. We've tested this method and it prooved to be robust to various failures. Still, if your system has high enough limits, and `file_descriptor` is a supported strategy, we do not recommend switching to this one.

CHAPTER 15

Legacy package - torch.legacy

Package containing code ported from Lua torch.

To make it possible to work with existing models and ease the transition for current Lua torch users, we've created this package. You can find the `nn` code in `torch.legacy.nn`, and `optim` in `torch.legacy.optim`. The APIs should exactly match Lua torch.

CHAPTER 16

torch.cuda

This package adds support for CUDA tensor types, that implement the same function as CPU tensors, but they utilize GPUs for computation.

It is lazily initialized, so you can always import it, and use *is_available()* to determine if your system supports CUDA.

CUDA semantics has more details about working with CUDA.

torch.cuda.**current_blas_handle**()
Returns cublasHandle_t pointer to current cuBLAS handle

torch.cuda.**current_device**()
Returns the index of a currently selected device.

torch.cuda.**current_stream**()
Returns a currently selected *Stream*.

class torch.cuda.**device**(*idx*)
Context-manager that changes the selected device.

idx (*int*) – device index to select. It's a no-op if this argument is negative.

torch.cuda.**device_count**()
Returns the number of GPUs available.

class torch.cuda.**device_of**(*obj*)
Context-manager that changes the current device to that of given object.

You can use both tensors and storages as arguments. If a given object is not allocated on a GPU, this is a no-op.

obj (*Tensor or Storage*) – object allocated on the selected device.

torch.cuda.**is_available**()
Returns a bool indicating if CUDA is currently available.

torch.cuda.**set_device**(*device*)
Sets the current device.

Usage of this function is discouraged in favor of *device*. In most cases it's better to use CUDA_VISIBLE_DEVICES environmental variable.

device (*int*) – selected device. This function is a no-op if this argument is negative.

torch.cuda.**stream**(**args*, ***kwds*)
Context-manager that selects a given stream.

All CUDA kernels queued within its context will be enqueued on a selected stream.

stream (Stream) – selected stream. This manager is a no-op if it's None.

torch.cuda.**synchronize**()
Waits for all kernels in all streams on current device to complete.

Communication collectives

torch.cuda.comm.**broadcast**(*tensor*, *devices*)
Broadcasts a tensor to a number of GPUs.

- **tensor** (Tensor) – tensor to broadcast.
- **devices** (*Iterable*) – an iterable of devices among which to broadcast. Note that it should be like (src, dst1, dst2, ...), the first element of which is the source device to broadcast from.

A tuple containing copies of the tensor, placed on devices corresponding to indices from devices.

torch.cuda.comm.**reduce_add**(*inputs*, *destination=None*)
Sums tensors from multiple GPUs.

All inputs should have matching shapes.

- **inputs** (*Iterable[Tensor]*) – an iterable of tensors to add.
- **destination** (*int, optional*) – a device on which the output will be placed (default: current device).

A tensor containing an elementwise sum of all inputs, placed on the destination device.

torch.cuda.comm.**scatter**(*tensor*, *devices*, *chunk_sizes=None*, *dim=0*, *streams=None*)
Scatters tensor across multiple GPUs.

- **tensor** (Tensor) – tensor to scatter.
- **devices** (*Iterable[int]*) – iterable of ints, specifying among which devices the tensor should be scattered.
- **chunk_sizes** (*Iterable[int], optional*) – sizes of chunks to be placed on each device. It should match devices in length and sum to tensor.size(dim). If not specified, the tensor will be divided into equal chunks.
- **dim** (*int, optional*) – A dimension along which to chunk the tensor.

A tuple containing chunks of the tensor, spread accross given devices.

torch.cuda.comm.**gather**(*tensors*, *dim=0*, *destination=None*)
Gathers tensors from multiple GPUs.

Tensor sizes in all dimension different than `dim` have to match.

- **tensors** (*Iterable[Tensor]*) – iterable of tensors to gather.
- **dim** (*int*) – a dimension along which the tensors will be concatenated.
- **destination** (*int, optional*) – output device (-1 means CPU, default: current device)

A tensor located on `destination` device, that is a result of concatenating `tensors` along `dim`.

Streams and events

class torch.cuda.**Stream**
Wrapper around a CUDA stream.

- **device** (*int, optional*) – a device on which to allocate the Stream.
- **priority** (*int, optional*) – priority of the stream. Lower numbers represent higher priorities.

query()
Checks if all the work submitted has been completed.

A boolean indicating if all kernels in this stream are completed.

record_event(*event=None*)
Records an event.

event (Event, *optional*) – event to record. If not given, a new one will be allocated.

Recorded event.

synchronize()
Wait for all the kernels in this stream to complete.

wait_event(*event*)
Makes all future work submitted to the stream wait for an event.

event (Event) – an event to wait for.

wait_stream(*stream*)
Synchronizes with another stream.

All future work submitted to this stream will wait until all kernels submitted to a given stream at the time of call complete.

stream (Stream) – a stream to synchronize.

class torch.cuda.**Event**(*enable_timing=False*, *blocking=False*, *interprocess=False*, *_handle=None*)
Wrapper around CUDA event.

- **enable_timing** (*bool*) – indicates if the event should measure time (default: False)
- **blocking** (*bool*) – if true, *wait()* will be blocking (default: False)

- **interprocess** (*bool*) – if true, the event can be shared between processes (default: False)

elapsed_time(*end_event*)
Returns the time elapsed before the event was recorded.

ipc_handle()
Returns an IPC handle of this event.

query()
Checks if the event has been recorded.

A boolean indicating if the event has been recorded.

record(*stream=None*)
Records the event in a given stream.

synchronize()
Synchronizes with the event.

wait(*stream=None*)
Makes a given stream wait for the event.

CHAPTER 17

torch.utils.ffi

CHAPTER 18

torch.utils.data

class torch.utils.data.**Dataset**
An abstract class representing a Dataset.

All other datasets should subclass it. All subclasses should override __len__, that provides the size of the dataset, and __getitem__, supporting integer indexing in range from 0 to len(self) exclusive.

class torch.utils.data.**TensorDataset**(*data_tensor*, *target_tensor*)
Dataset wrapping data and target tensors.

Each sample will be retrieved by indexing both tensors along the first dimension.

- **data_tensor** (Tensor) – contains sample data.
- **target_tensor** (Tensor) – contains sample targets (labels).

class torch.utils.data.**DataLoader**(*dataset*, *batch_size=1*, *shuffle=False*, *sampler=None*, *num_workers=0*, *collate_fn=<function default_collate>*, *pin_memory=False*)
Data loader. Combines a dataset and a sampler, and provides single- or multi-process iterators over the dataset.

- **dataset** (Dataset) – dataset from which to load the data.
- **batch_size** (*int, optional*) – how many samples per batch to load (default: 1).
- **shuffle** (*bool, optional*) – set to True to have the data reshuffled at every epoch (default: False).
- **sampler** (Sampler, *optional*) – defines the strategy to draw samples from the dataset. If specified, the shuffle argument is ignored.
- **num_workers** (*int, optional*) – how many subprocesses to use for data loading. 0 means that the data will be loaded in the main process (default: 0)
- **collate_fn** (*callable, optional*) –
- **pin_memory** (*bool, optional*) –

class torch.utils.data.sampler.**Sampler**(*data_source*)
Base class for all Samplers.

Every Sampler subclass has to provide an __iter__ method, providing a way to iterate over indices of dataset elements, and a __len__ method that returns the length of the returned iterators.

class torch.utils.data.sampler.**SequentialSampler**(*data_source*)
Samples elements sequentially, always in the same order.

data_source (Dataset) – dataset to sample from

class torch.utils.data.sampler.**RandomSampler**(*data_source*)
Samples elements randomly, without replacement.

data_source (Dataset) – dataset to sample from

class torch.utils.data.sampler.**SubsetRandomSampler**(*indices*)
Samples elements randomly from a given list of indices, without replacement.

indices (*list*) – a list of indices

class torch.utils.data.sampler.**WeightedRandomSampler**(*weights*, *num_samples*, *replacement=True*)
Samples elements from [0,..,len(weights)-1] with given probabilities (weights). :param weights: a list of weights, not necessary summing up to one :type weights: list :param num_samples: number of samples to draw :type num_samples: int

CHAPTER 19

torch.utils.model_zoo

torch.utils.model_zoo.**load_url**(*url*, *model_dir=None*)

Loads the Torch serialized object at the given URL.

If the object is already present in *model_dir*, it's deserialied and returned. The filename part of the URL should follow the naming convention `filename-<sha256>.ext` where `<sha256>` is the first eight or more digits of the SHA256 hash of the contents of the file. The hash is used to ensure unique names and to verify the contents of the file.

The default value of *model_dir* is `$TORCH_HOME/models` where `$TORCH_HOME` defaults to `~/.torch`. The default directory can be overriden with the `$TORCH_MODEL_ZOO` environement variable.

- **url** (*string*) – URL of the object to download
- **model_dir** (*string, optional*) – directory in which to save the object

Example

```
>>> state_dict = torch.utils.model_zoo.load_url('https://s3.amazonaws.com/pytorch/
↪models/resnet18-5c106cde.pth')
```

CHAPTER 20

torchvision

The `torchvision` package consists of popular datasets, model architectures, and common image transformations for computer vision.

CHAPTER 21

torchvision.datasets

The following dataset loaders are available:

- *MNIST*
- *COCO (Captioning and Detection)*
- *LSUN Classification*
- *ImageFolder*
- *Imagenet-12*
- *CIFAR10 and CIFAR100*
- *STL10*

Datasets have the API:

- `__getitem__`
- `__len__` They all subclass from `torch.utils.data.Dataset` Hence, they can all be multi-threaded (python multiprocessing) using standard torch.utils.data.DataLoader.

For example:

```
torch.utils.data.DataLoader(coco_cap, batch_size=args.batchSize, shuffle=True,
num_workers=args.nThreads)
```

In the constructor, each dataset has a slightly different API as needed, but they all take the keyword args:

- `transform` - a function that takes in an image and returns a transformed version
- common stuff like `ToTensor`, `RandomCrop`, etc. These can be composed together with `transforms.Compose` (see transforms section below)
- `target_transform` - a function that takes in the target and transforms it. For example, take in the caption string and return a tensor of word indices.

MNIST

`dset.MNIST(root, train=True, transform=None, target_transform=None, download=False)`

- `root` : root directory of dataset where `processed/training.pt` and `processed/test.pt` exist.
- `train` : `True` = Training set, `False` = Test set
- `download` : `True` = downloads the dataset from the internet and puts it in root directory. If dataset already downloaded, place the processed dataset (function available in mnist.py) in the `processed` folder.

COCO

This requires the COCO API to be installed

Captions:

`dset.CocoCaptions(root="dir where images are", annFile="json annotation file", [transform, target_transform])`

Example:

```
import torchvision.datasets as dset
import torchvision.transforms as transforms
cap = dset.CocoCaptions(root = 'dir where images are',
                        annFile = 'json annotation file',
                        transform=transforms.ToTensor())

print('Number of samples: ', len(cap))
img, target = cap[3] # load 4th sample

print("Image Size: ", img.size())
print(target)
```

Output:

```
Number of samples: 82783
Image Size: (3L, 427L, 640L)
[u'A plane emitting smoke stream flying over a mountain.',
u'A plane darts across a bright blue sky behind a mountain covered in snow',
u'A plane leaves a contrail above the snowy mountain top.',
u'A mountain that has a plane flying overheard in the distance.',
u'A mountain view with a plume of smoke in the background']
```

Detection:

`dset.CocoDetection(root="dir where images are", annFile="json annotation file", [transform, target_transform])`

LSUN

`dset.LSUN(db_path, classes='train', [transform, target_transform])`

- db_path = root directory for the database files
- `classes` = `'train'` (all categories, training set), `'val'` (all categories, validation set), `'test'` (all categories, test set)
- [`'bedroom\_train'`, `'church\_train'`, ...] : a list of categories to load

ImageFolder

A generic data loader where the images are arranged in this way:

```
root/dog/xxx.png
root/dog/xxy.png
root/dog/xxz.png

root/cat/123.png
root/cat/nsdf3.png
root/cat/asd932_.png
```

`dset.ImageFolder(root="root folder path", [transform, target_transform])`

It has the members:

- `self.classes` - The class names as a list
- `self.class_to_idx` - Corresponding class indices
- `self.imgs` - The list of (image path, class-index) tuples

Imagenet-12

This is simply implemented with an ImageFolder dataset.

The data is preprocessed as described here

Here is an example.

CIFAR

`dset.CIFAR10(root, train=True, transform=None, target_transform=None, download=False)`

`dset.CIFAR100(root, train=True, transform=None, target_transform=None, download=False)`

- `root` : root directory of dataset where there is folder `cifar-10-batches-py`
- `train` : `True` = Training set, `False` = Test set
- `download` : `True` = downloads the dataset from the internet and puts it in root directory. If dataset already downloaded, doesn't do anything.

STL10

```
dset.STL10(root, split='train', transform=None, target_transform=None,
download=False)
```

- `root` : root directory of dataset where there is folder `stl10_binary`
- `split` : `'train'` = Training set, `'test'` = Test set, `'unlabeled'` = Unlabeled set, `'train+unlabeled'` = Training + Unlabeled set (missing label marked as `-1`)
- `download` : `True` = downloads the dataset from the internet and puts it in root directory. If dataset already downloaded, doesn't do anything.

CHAPTER 22

torchvision.models

The models subpackage contains definitions for the following model architectures:

- AlexNet
- VGG
- ResNet
- SqueezeNet
- DenseNet

You can construct a model with random weights by calling its constructor:

```
import torchvision.models as models
resnet18 = models.resnet18()
alexnet = models.alexnet()
squeezenet = models.squeezenet1_0()
densenet = models.densenet_161()
```

We provide pre-trained models for the ResNet variants and AlexNet, using the PyTorch *torch.utils.model_zoo*. These can constructed by passing `pretrained=True`:

```
import torchvision.models as models
resnet18 = models.resnet18(pretrained=True)
alexnet = models.alexnet(pretrained=True)
```

ImageNet 1-crop error rates (224x224)

Network	Top-1 error	Top-5 error
ResNet-18	30.24	10.92
ResNet-34	26.70	8.58
ResNet-50	23.85	7.13
ResNet-101	22.63	6.44
ResNet-152	21.69	5.94
Inception v3	22.55	6.44
AlexNet	43.45	20.91
VGG-11	30.98	11.37
VGG-13	30.07	10.75
VGG-16	28.41	9.62
VGG-19	27.62	9.12
SqueezeNet 1.0	41.90	19.58
SqueezeNet 1.1	41.81	19.38
Densenet-121	25.35	7.83
Densenet-169	24.00	7.00
Densenet-201	22.80	6.43
Densenet-161	22.35	6.20

`torchvision.models.`**`alexnet`**(*pretrained=False*, ***kwargs*)
AlexNet model architecture from the "One weird trick..." paper.

`pretrained` (*bool*) – If True, returns a model pre-trained on ImageNet

`torchvision.models.`**`resnet18`**(*pretrained=False*, ***kwargs*)
Constructs a ResNet-18 model.

`pretrained` (*bool*) – If True, returns a model pre-trained on ImageNet

`torchvision.models.`**`resnet34`**(*pretrained=False*, ***kwargs*)
Constructs a ResNet-34 model.

`pretrained` (*bool*) – If True, returns a model pre-trained on ImageNet

`torchvision.models.`**`resnet50`**(*pretrained=False*, ***kwargs*)
Constructs a ResNet-50 model.

`pretrained` (*bool*) – If True, returns a model pre-trained on ImageNet

`torchvision.models.`**`resnet101`**(*pretrained=False*, ***kwargs*)
Constructs a ResNet-101 model.

`pretrained` (*bool*) – If True, returns a model pre-trained on ImageNet

`torchvision.models.`**`resnet152`**(*pretrained=False*, ***kwargs*)
Constructs a ResNet-152 model.

`pretrained` (*bool*) – If True, returns a model pre-trained on ImageNet

`torchvision.models.`**`vgg11`**(*pretrained=False*, ***kwargs*)
VGG 11-layer model (configuration "A")

`pretrained` (*bool*) – If True, returns a model pre-trained on ImageNet

`torchvision.models.`**`vgg11_bn`**(***kwargs*)
VGG 11-layer model (configuration "A") with batch normalization

`torchvision.models.`**`vgg13`**(*pretrained=False*, ***kwargs*)
VGG 13-layer model (configuration "B")

`pretrained` (*bool*) – If True, returns a model pre-trained on ImageNet

torchvision.models.**vgg13_bn**(**kwargs)
VGG 13-layer model (configuration "B") with batch normalization

torchvision.models.**vgg16**(*pretrained=False*, **kwargs)
VGG 16-layer model (configuration "D")

pretrained (*bool*) – If True, returns a model pre-trained on ImageNet

torchvision.models.**vgg16_bn**(**kwargs)
VGG 16-layer model (configuration "D") with batch normalization

torchvision.models.**vgg19**(*pretrained=False*, **kwargs)
VGG 19-layer model (configuration "E")

pretrained (*bool*) – If True, returns a model pre-trained on ImageNet

torchvision.models.**vgg19_bn**(**kwargs)
VGG 19-layer model (configuration 'E') with batch normalization

CHAPTER 23

torchvision.transforms

class `torchvision.transforms.`**`Compose`**(*transforms*)
Composes several transforms together.

`transforms` (*List[Transform]*) – list of transforms to compose.

Example

```
>>> transforms.Compose([
>>>     transforms.CenterCrop(10),
>>>     transforms.ToTensor(),
>>> ])
```

Transforms on PIL.Image

class `torchvision.transforms.`**`Scale`**(*size*, *interpolation=2*)
Rescales the input PIL.Image to the given 'size'. 'size' will be the size of the smaller edge. For example, if height > width, then image will be rescaled to (size * height / width, size) size: size of the smaller edge interpolation: Default: PIL.Image.BILINEAR

class `torchvision.transforms.`**`CenterCrop`**(*size*)
Crops the given PIL.Image at the center to have a region of the given size. size can be a tuple (target_height, target_width) or an integer, in which case the target will be of a square shape (size, size)

class `torchvision.transforms.`**`RandomCrop`**(*size*, *padding=0*)
Crops the given PIL.Image at a random location to have a region of the given size. size can be a tuple (target_height, target_width) or an integer, in which case the target will be of a square shape (size, size)

class `torchvision.transforms.`**`RandomHorizontalFlip`**
Randomly horizontally flips the given PIL.Image with a probability of 0.5

class torchvision.transforms.**RandomSizedCrop**(*size*, *interpolation=2*)
Random crop the given PIL.Image to a random size of (0.08 to 1.0) of the original size and and a random aspect ratio of 3/4 to 4/3 of the original aspect ratio This is popularly used to train the Inception networks size: size of the smaller edge interpolation: Default: PIL.Image.BILINEAR

class torchvision.transforms.**Pad**(*padding*, *fill=0*)
Pads the given PIL.Image on all sides with the given "pad" value

Transforms on torch.*Tensor

class torchvision.transforms.**Normalize**(*mean*, *std*)
Given mean: (R, G, B) and std: (R, G, B), will normalize each channel of the torch.*Tensor, i.e. channel = (channel - mean) / std

Conversion Transforms

class torchvision.transforms.**ToTensor**
Converts a PIL.Image or numpy.ndarray (H x W x C) in the range [0, 255] to a torch.FloatTensor of shape (C x H x W) in the range [0.0, 1.0].

class torchvision.transforms.**ToPILImage**
Converts a torch.*Tensor of shape C x H x W or a numpy ndarray of shape H x W x C to a PIL.Image while preserving value range.

Generic Transforms

class torchvision.transforms.**Lambda**(*lambd*)
Applies a lambda as a transform.

CHAPTER 24

torchvision.utils

torchvision.utils.**make_grid**(*tensor*, *nrow=8*, *padding=2*, *normalize=False*, *range=None*, *scale_each=False*)

Given a 4D mini-batch Tensor of shape (B x C x H x W), or a list of images all of the same size, makes a grid of images of size (B / nrow, nrow).

normalize=True will shift the image to the range (0, 1), by subtracting the minimum and dividing by the maximum pixel value.

if range=(min, max) where min and max are numbers, then these numbers are used to normalize the image.

scale_each=True will scale each image in the batch of images separately rather than computing the (min, max) over all images.

[Example usage is given in this notebook](https://gist.github.com/anonymous/bf16430f7750c023141c562f3e9f2a91)

torchvision.utils.**save_image**(*tensor*, *filename*, *nrow=8*, *padding=2*, *normalize=False*, *range=None*, *scale_each=False*)

Saves a given Tensor into an image file. If given a mini-batch tensor, will save the tensor as a grid of images by calling *make_grid*. All options after *filename* are passed through to *make_grid*. Refer to it's documentation for more details

CHAPTER 25

Indices and tables

- genindex
- modindex

t

T

U

V

W

Z