PyTorch Documentation

Release master

Torch Contributors

Oct 18, 2019

NOTES

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

CHAPTER

ONE

AUTOGRAD MECHANICS

This note will present an overview of how autograd works and records the operations. Its 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.

1.1 Excluding subgraphs from backward

Every Tensor has a flag: `requires_grad` that allows for fine grained exclusion of subgraphs from gradient computation and can increase efficiency.

1.1.1 `requires_grad`

If theres a single input to an operation that requires gradient, its output will also require gradient. Conversely, only if all inputs dont require gradient, the output also wont require it. Backward computation is never performed in the subgraphs, where all Tensors didnt require gradients.

```
>>> x = torch.randn(5, 5)  # requires_grad=False by default
>>> y = torch.randn(5, 5)  # requires_grad=False by default
>>> z = 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 youre not going to use gradients w.r.t. some parameters. For example if you want to finetune a pretrained CNN, its 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)
```

1.2 How autograd encodes the history

Autograd is reverse automatic differentiation system. Conceptually, autograd records a graph recording all of the operations that created the data as you execute operations, giving you a directed acyclic graph whose leaves are the input tensors and roots are the output tensors. By tracing this graph from roots to leaves, you can automatically compute the gradients using the chain rule.

Internally, autograd represents this graph as a graph of `Function` objects (really expressions), which can be `apply()` ed to compute the result of evaluating the graph. When computing the forwards pass, autograd simultaneously performs the requested computations and builds up a graph representing the function that computes the gradient (the `.grad_fn` attribute of each *`torch.Tensor`* is an entry point into this graph). When the forwards pass is completed, we evaluate this graph in the backwards pass to 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 dont have to encode all possible paths before you launch the training - what you run is what you differentiate.

1.3 In-place operations with autograd

Supporting in-place operations in autograd is a hard matter, and we discourage their use in most cases. Autograds 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 youre 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. In-place operations can potentially overwrite values required to compute gradients.
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 Tensors 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 `Tensor`.

1.4 In-place correctness checks

Every tensor keeps a version counter, that is incremented every time it is marked dirty in any operation. When a Function saves any tensors for backward, a version counter of their containing Tensor is saved as well. Once you access `self.saved_tensors` it is checked, and if it is greater than the saved value an error is raised. This ensures that if youre using in-place functions and not seeing any errors, you can be sure that the computed gradients are correct.

CHAPTER

TWO

BROADCASTING SEMANTICS

Many PyTorch operations support NumPy Broadcasting Semantics.

In short, if a PyTorch operation supports broadcast, then its Tensor arguments can be automatically expanded to be of equal sizes (without making copies of the data).

2.1 General semantics

Two tensors are broadcastable if the following rules hold:

- Each tensor has at least one dimension.
- When iterating over the dimension sizes, starting at the trailing dimension, the dimension sizes must either be equal, one of them is 1, or one of them does not exist.

For Example:

```
>>> x=torch.empty(5,7,3)
>>> y=torch.empty(5,7,3)
# same shapes are always broadcastable (i.e. the above rules always hold)

>>> x=torch.empty((0,))
>>> y=torch.empty(2,2)
# x and y are not broadcastable, because x does not have at least 1 dimension

# can line up trailing dimensions
>>> x=torch.empty(5,3,4,1)
>>> y=torch.empty(  3,1,1)
# x and y are broadcastable.
# 1st trailing dimension: both have size 1
# 2nd trailing dimension: y has size 1
# 3rd trailing dimension: x size == y size
# 4th trailing dimension: y dimension doesn't exist

# but:
>>> x=torch.empty(5,2,4,1)
>>> y=torch.empty(  3,1,1)
# x and y are not broadcastable, because in the 3rd trailing dimension 2 != 3
```

If two tensors `x`, `y` are broadcastable, the resulting tensor size is calculated as follows:

- If the number of dimensions of `x` and `y` are not equal, prepend 1 to the dimensions of the tensor with fewer dimensions to make them equal length.

- Then, for each dimension size, the resulting dimension size is the max of the sizes of `x` and `y` along that dimension.

For Example:

```
# can line up trailing dimensions to make reading easier
>>> x=torch.empty(5,1,4,1)
>>> y=torch.empty(  3,1,1)
>>> (x+y).size()
torch.Size([5, 3, 4, 1])

# but not necessary:
>>> x=torch.empty(1)
>>> y=torch.empty(3,1,7)
>>> (x+y).size()
torch.Size([3, 1, 7])

>>> x=torch.empty(5,2,4,1)
>>> y=torch.empty(3,1,1)
>>> (x+y).size()
RuntimeError: The size of tensor a (2) must match the size of tensor b (3) at non-
↪singleton dimension 1
```

2.2 In-place semantics

One complication is that in-place operations do not allow the in-place tensor to change shape as a result of the broadcast.

For Example:

```
>>> x=torch.empty(5,3,4,1)
>>> y=torch.empty(3,1,1)
>>> (x.add_(y)).size()
torch.Size([5, 3, 4, 1])

# but:
>>> x=torch.empty(1,3,1)
>>> y=torch.empty(3,1,7)
>>> (x.add_(y)).size()
RuntimeError: The expanded size of the tensor (1) must match the existing size (7) at␣
↪non-singleton dimension 2.
```

2.3 Backwards compatibility

Prior versions of PyTorch allowed certain pointwise functions to execute on tensors with different shapes, as long as the number of elements in each tensor was equal. The pointwise operation would then be carried out by viewing each tensor as 1-dimensional. PyTorch now supports broadcasting and the 1-dimensional pointwise behavior is considered deprecated and will generate a Python warning in cases where tensors are not broadcastable, but have the same number of elements.

Note that the introduction of broadcasting can cause backwards incompatible changes in the case where two tensors do not have the same shape, but are broadcastable and have the same number of elements. For Example:

```
>>> torch.add(torch.ones(4,1), torch.randn(4))
```

would previously produce a Tensor with size: torch.Size([4,1]), but now produces a Tensor with size: torch.Size([4,4]). In order to help identify cases in your code where backwards incompatibilities introduced by broadcasting may exist, you may set *torch.utils.backcompat.broadcast_warning.enabled* to *True*, which will generate a python warning in such cases.

For Example:

```
>>> torch.utils.backcompat.broadcast_warning.enabled=True
>>> torch.add(torch.ones(4,1), torch.ones(4))
__main__:1: UserWarning: self and other do not have the same shape, but are␣
↪broadcastable, and have the same number of elements.
Changing behavior in a backwards incompatible manner to broadcasting rather than␣
↪viewing as 1-dimensional.
```

CHAPTER

THREE

CPU THREADING AND TORCHSCRIPT INFERENCE

PyTorch allows using multiple CPU threads during TorchScript model inference. The following figure shows different levels of parallelism one would find in a typical application:

One or more inference threads execute a models forward pass on the given inputs. Each inference thread invokes a JIT interpreter that executes the ops of a model inline, one by one. A model can utilize a `fork` TorchScript primitive to launch an asynchronous task. Forking several operations at once results in a task that is executed in parallel. The `fork` operator returns a `future` object which can be used to synchronize on later, for example:

```
@torch.jit.script
def compute_z(x):
    return torch.mm(x, self.w_z)
```

(continues on next page)

(continued from previous page)

```
@torch.jit.script
def forward(x):
    # launch compute_z asynchronously:
    fut = torch.jit._fork(compute_z, x)
    # execute the next operation in parallel to compute_z:
    y = torch.mm(x, self.w_y)
    # wait for the result of compute_z:
    z = torch.jit._wait(fut)
    return y + z
```

PyTorch uses a single thread pool for the inter-op parallelism, this thread pool is shared by all inference tasks that are forked within the application process.

In addition to the inter-op parallelism, PyTorch can also utilize multiple threads within the ops (*intra-op parallelism*). This can be useful in many cases, including element-wise ops on large tensors, convolutions, GEMMs, embedding lookups and others.

3.1 Build options

PyTorch uses an internal ATen library to implement ops. In addition to that, PyTorch can also be built with support of external libraries, such as MKL and MKL-DNN, to speed up computations on CPU.

ATen, MKL and MKL-DNN support intra-op parallelism and depend on the following parallelization libraries to implement it:

- OpenMP - a standard (and a library, usually shipped with a compiler), widely used in external libraries;
- TBB - a newer parallelization library optimized for task-based parallelism and concurrent environments.

OpenMP historically has been used by a large number of libraries. It is known for a relative ease of use and support for loop-based parallelism and other primitives. At the same time OpenMP is not known for a good interoperability with other threading libraries used by the application. In particular, OpenMP does not guarantee that a single per-process intra-op thread pool is going to be used in the application. On the contrary, two different inter-op threads will likely use different OpenMP thread pools for intra-op work. This might result in a large number of threads used by the application.

TBB is used to a lesser extent in external libraries, but, at the same time, is optimized for the concurrent environments. PyTorchs TBB backend guarantees that theres a separate, single, per-process intra-op thread pool used by all of the ops running in the application.

Depending of the use case, one might find one or another parallelization library a better choice in their application.

PyTorch allows selecting of the parallelization backend used by ATen and other libraries at the build time with the following build options:

Library	Build Option	Values	Notes
ATen	`ATEN_THREADING`	`OMP` (default), `TBB`	
MKL	`MKL_THREADING`	(same)	To enable MKL use `BLAS=MKL`
MKL-DNN	`MKLDNN_THREADING`	(same)	To enable MKL-DNN use `USE_MKLDNN=1`

It is strongly recommended not to mix OpenMP and TBB within one build.

Any of the `TBB` values above require `USE_TBB=1` build setting (default: OFF). A separate setting `USE_OPENMP=1` (default: ON) is required for OpenMP parallelism.

3.2 Runtime API

The following API is used to control thread settings:

Type of parallelism	Settings	Notes
Inter-op parallelism	`at::set_num_interop_threads`, `at::get_num_interop_threads` (C++) `set_num_interop_threads`, `get_num_interop_threads` (Python, *torch* module)	`set*` functions can only be called once and only during the startup, before the actual operators running; Default number of threads: number of CPU cores.
Intra-op parallelism	`at::set_num_threads`, `at::get_num_threads` (C++) `set_num_threads`, `get_num_threads` (Python, *torch* module) Environment variables: `OMP_NUM_THREADS` and `MKL_NUM_THREADS`	

For the intra-op parallelism settings, `at::set_num_threads`, `torch.set_num_threads` always take precedence over environment variables, `MKL_NUM_THREADS` variable takes precedence over `OMP_NUM_THREADS`.

Note: `parallel_info` utility prints information about thread settings and can be used for debugging. Similar output can be also obtained in Python with `torch.__config__.parallel_info()` call.

CHAPTER

FOUR

CUDA SEMANTICS

torch.cuda is used to set up and run CUDA operations. It keeps track of the currently selected GPU, and all CUDA tensors you allocate will by default be created on that device. 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 irrespective of the 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 exception of *copy_()* and other methods with copy-like functionality such as *to()* and *cuda()*. Unless you enable peer-to-peer memory access, any attempts to launch ops on tensors spread across different devices will raise an error.

Below you can find a small example showcasing this:

```
cuda = torch.device('cuda')     # Default CUDA device
cuda0 = torch.device('cuda:0')
cuda2 = torch.device('cuda:2')  # GPU 2 (these are 0-indexed)

x = torch.tensor([1., 2.], device=cuda0)
# x.device is device(type='cuda', index=0)
y = torch.tensor([1., 2.]).cuda()
# y.device is device(type='cuda', index=0)

with torch.cuda.device(1):
    # allocates a tensor on GPU 1
    a = torch.tensor([1., 2.], device=cuda)

    # transfers a tensor from CPU to GPU 1
    b = torch.tensor([1., 2.]).cuda()
    # a.device and b.device are device(type='cuda', index=1)

    # You can also use ``Tensor.to`` to transfer a tensor:
    b2 = torch.tensor([1., 2.]).to(device=cuda)
    # b.device and b2.device are device(type='cuda', index=1)

    c = a + b
    # c.device is device(type='cuda', index=1)

    z = x + y
    # z.device is device(type='cuda', index=0)

    # even within a context, you can specify the device
    # (or give a GPU index to the .cuda call)
    d = torch.randn(2, device=cuda2)
    e = torch.randn(2).to(cuda2)
```

(continues on next page)

(continued from previous page)

```
    f = torch.randn(2).cuda(cuda2)
    # d.device, e.device, and f.device are all device(type='cuda', index=2)
```

4.1 Asynchronous execution

By default, GPU operations are asynchronous. When you call a function that uses the GPU, the operations are *enqueued* to the particular device, but not necessarily executed until later. This allows us to execute more computations in parallel, including operations on CPU or other GPUs.

In general, the effect of asynchronous computation is invisible to the caller, because (1) each device executes operations in the order they are queued, and (2) PyTorch automatically performs necessary synchronization when copying data between CPU and GPU or between two GPUs. Hence, computation will proceed as if every operation was executed synchronously.

You can force synchronous computation by setting environment variable `CUDA_LAUNCH_BLOCKING=1`. This can be handy when an error occurs on the GPU. (With asynchronous execution, such an error isnt reported until after the operation is actually executed, so the stack trace does not show where it was requested.)

A consequence of the asynchronous computation is that time measurements without synchronizations are not accurate. To get precise measurements, one should either call *torch.cuda.synchronize()* before measuring, or use *torch.cuda.Event* to record times as following:

```
start_event = torch.cuda.Event(enable_timing=True)
end_event = torch.cuda.Event(enable_timing=True)
start_event.record()

# Run some things here

end_event.record()
torch.cuda.synchronize()  # Wait for the events to be recorded!
elapsed_time_ms = start_event.elapsed_time(end_event)
```

As an exception, several functions such as *to()* and *copy_()* admit an explicit `non_blocking` argument, which lets the caller bypass synchronization when it is unnecessary. Another exception is CUDA streams, explained below.

4.1.1 CUDA streams

A CUDA stream is a linear sequence of execution that belongs to a specific device. You normally do not need to create one explicitly: by default, each device uses its own default stream.

Operations inside each stream are serialized in the order they are created, but operations from different streams can execute concurrently in any relative order, unless explicit synchronization functions (such as *synchronize()* or *wait_stream()*) are used. For example, the following code is incorrect:

```
cuda = torch.device('cuda')
s = torch.cuda.Stream()  # Create a new stream.
A = torch.empty((100, 100), device=cuda).normal_(0.0, 1.0)
with torch.cuda.stream(s):
    # sum() may start execution before normal_() finishes!
    B = torch.sum(A)
```

When the current stream is the default stream, PyTorch automatically performs necessary synchronization when data is moved around, as explained above. However, when using non-default streams, it is the users responsibility to ensure proper synchronization.

4.2 Memory management

PyTorch uses a caching memory allocator to speed up memory allocations. This allows fast memory deallocation without device synchronizations. However, the unused memory managed by the allocator will still show as if used in `nvidia-smi`. You can use *`memory_allocated()`* and *`max_memory_allocated()`* to monitor memory occupied by tensors, and use `memory_reserved()` and `max_memory_reserved()` to monitor the total amount of memory managed by the caching allocator. Calling *`empty_cache()`* releases all **unused** cached memory from PyTorch so that those can be used by other GPU applications. However, the occupied GPU memory by tensors will not be freed so it can not increase the amount of GPU memory available for PyTorch.

For more advanced users, we offer more comprehensive memory benchmarking via `memory_stats()`. We also offer the capability to capture a complete snapshot of the memory allocator state via `memory_snapshot()`, which can help you understand the underlying allocation patterns produced by your code.

4.3 cuFFT plan cache

For each CUDA device, an LRU cache of cuFFT plans is used to speed up repeatedly running FFT methods (e.g., *`torch.fft()`*) on CUDA tensors of same geometry with same configuration. Because some cuFFT plans may allocate GPU memory, these caches have a maximum capacity.

You may control and query the properties of the cache of current device with the following APIs:

- `torch.backends.cuda.cufft_plan_cache.max_size` gives the capacity of the cache (default is 4096 on CUDA 10 and newer, and 1023 on older CUDA versions). Setting this value directly modifies the capacity.
- `torch.backends.cuda.cufft_plan_cache.size` gives the number of plans currently residing in the cache.
- `torch.backends.cuda.cufft_plan_cache.clear()` clears the cache.

To control and query plan caches of a non-default device, you can index the `torch.backends.cuda.cufft_plan_cache` object with either a `torch.device` object or a device index, and access one of the above attributes. E.g., to set the capacity of the cache for device `1`, one can write `torch.backends.cuda.cufft_plan_cache[1].max_size = 10`.

4.4 Best practices

4.4.1 Device-agnostic code

Due to the structure of PyTorch, you may need to explicitly write device-agnostic (CPU or GPU) code; an example may be creating a new tensor as the initial hidden state of a recurrent neural network.

The first step is to determine whether the GPU should be used or not. A common pattern is to use Pythons `argparse` module to read in user arguments, and have a flag that can be used to disable CUDA, in combination with *`is_available()`*. In the following, `args.device` results in a `torch.device` object that can be used to move tensors to CPU or CUDA.

```
import argparse
import torch

parser = argparse.ArgumentParser(description='PyTorch Example')
parser.add_argument('--disable-cuda', action='store_true',
```

(continues on next page)

(continued from previous page)

```
                    help='Disable CUDA')
args = parser.parse_args()
args.device = None
if not args.disable_cuda and torch.cuda.is_available():
    args.device = torch.device('cuda')
else:
    args.device = torch.device('cpu')
```

Now that we have `args.device`, we can use it to create a Tensor on the desired device.

```
x = torch.empty((8, 42), device=args.device)
net = Network().to(device=args.device)
```

This can be used in a number of cases to produce device agnostic code. Below is an example when using a dataloader:

```
cuda0 = torch.device('cuda:0')  # CUDA GPU 0
for i, x in enumerate(train_loader):
    x = x.to(cuda0)
```

When working with multiple GPUs on a system, you can use the `CUDA_VISIBLE_DEVICES` environment flag to manage which GPUs are available to PyTorch. As mentioned above, to manually control which GPU a tensor is created on, the best practice is to use a *`torch.cuda.device`* context manager.

```
print("Outside device is 0")  # On device 0 (default in most scenarios)
with torch.cuda.device(1):
    print("Inside device is 1")  # On device 1
print("Outside device is still 0")  # On device 0
```

If you have a tensor and would like to create a new tensor of the same type on the same device, then you can use a `torch.Tensor.new_*` method (see *`torch.Tensor`*). Whilst the previously mentioned `torch.*` factory functions (*Creation Ops*) depend on the current GPU context and the attributes arguments you pass in, `torch.Tensor.new_*` methods preserve the device and other attributes of the tensor.

This is the recommended practice when creating modules in which new tensors need to be created internally during the forward pass.

```
cuda = torch.device('cuda')
x_cpu = torch.empty(2)
x_gpu = torch.empty(2, device=cuda)
x_cpu_long = torch.empty(2, dtype=torch.int64)

y_cpu = x_cpu.new_full([3, 2], fill_value=0.3)
print(y_cpu)

    tensor([[ 0.3000,  0.3000],
            [ 0.3000,  0.3000],
            [ 0.3000,  0.3000]])

y_gpu = x_gpu.new_full([3, 2], fill_value=-5)
print(y_gpu)

    tensor([[-5.0000, -5.0000],
            [-5.0000, -5.0000],
            [-5.0000, -5.0000]], device='cuda:0')

y_cpu_long = x_cpu_long.new_tensor([[1, 2, 3]])
```

(continues on next page)

(continued from previous page)

```
print(y_cpu_long)

    tensor([[ 1,  2,  3]])
```

If you want to create a tensor of the same type and size of another tensor, and fill it with either ones or zeros, *ones_like()* or *zeros_like()* are provided as convenient helper functions (which also preserve `torch.device` and `torch.dtype` of a Tensor).

```
x_cpu = torch.empty(2, 3)
x_gpu = torch.empty(2, 3)

y_cpu = torch.ones_like(x_cpu)
y_gpu = torch.zeros_like(x_gpu)
```

4.4.2 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 `non_blocking=True` argument to a *to()* or 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.

4.4.3 Use nn.DataParallel instead of multiprocessing

Most use cases involving batched inputs 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

FIVE

EXTENDING PYTORCH

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

5.1 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 2 methods:

- *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. All kinds of Python objects are accepted here. `Tensor` arguments that track history (i.e., with `requires_grad=True`) will be converted to ones that dont track history before the call, and their use will be registered in the graph. Note that this logic wont traverse lists/dicts/any other data structures and will only consider `Tensor` s that are direct arguments to the call. You can return either a single `Tensor` output, or a `tuple` of `Tensor` s if there are multiple outputs. 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 `Tensor` 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. its corresponding input. If your inputs didnt require gradient (`needs_input_grad` is a tuple of booleans indicating whether each input needs gradient computation), or were non-`Tensor` objects, you can return `None`. Also, if you have optional arguments to *forward()* you can return more gradients than there were inputs, as long as theyre all `None`.

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

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

    # Note that both forward and backward are @staticmethods
    @staticmethod
    # bias is an optional argument
    def forward(ctx, input, weight, bias=None):
        ctx.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
    @staticmethod
    def backward(ctx, grad_output):
```

(continues on next page)

(continued from previous page)

```
        # 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 = ctx.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 ctx.needs_input_grad[0]:
            grad_input = grad_output.mm(weight)
        if ctx.needs_input_grad[1]:
            grad_weight = grad_output.t().mm(input)
        if bias is not None and ctx.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 aliasing their `apply` method:

```
linear = LinearFunction.apply
```

Here, we give an additional example of a function that is parametrized by non-Tensor arguments:

```
class MulConstant(Function):
    @staticmethod
    def forward(ctx, tensor, constant):
        # ctx is a context object that can be used to stash information
        # for backward computation
        ctx.constant = constant
        return tensor * constant

    @staticmethod
    def backward(ctx, grad_output):
        # We return as many input gradients as there were arguments.
        # Gradients of non-Tensor arguments to forward must be None.
        return grad_output * ctx.constant, None
```

Note: Inputs to `backward`, i.e., `grad_output`, can also be Tensors that track history. So if `backward` is implemented with differentiable operations, (e.g., invocation of another custom `function`), higher order derivatives will work.

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

# gradcheck takes a tuple of tensors 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 = (torch.randn(20,20,dtype=torch.double,requires_grad=True), torch.randn(30,20,
↪dtype=torch.double,requires_grad=True))
```

(continues on next page)

(continued from previous page)

```
test = gradcheck(linear, input, eps=1e-6, atol=1e-4)
print(test)
```

See *Numerical gradient checking* for more details on finite-difference gradient comparisons.

5.2 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.

5.2.1 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 lets assume that we want to implement a `Linear` module and we have the function implemented as in the listing above. Theres 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. Its 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):
        super(Linear, self).__init__()
        self.input_features = input_features
        self.output_features = output_features

        # nn.Parameter is a special kind of Tensor, 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 require gradients by default.
        self.weight = nn.Parameter(torch.Tensor(output_features, input_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)
```

(continues on next page)

(continued from previous page)

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

    def extra_repr(self):
        # (Optional)Set the extra information about this module. You can test
        # it by printing an object of this class.
        return 'in_features={}, out_features={}, bias={}'.format(
            self.in_features, self.out_features, self.bias is not None
        )
```

5.3 Writing custom C++ extensions

See this PyTorch tutorial for a detailed explanation and examples.

Documentations are available at *torch.utils.cpp_extension*.

5.4 Writing custom C extensions

Example available at this GitHub repository.

CHAPTER

SIX

FREQUENTLY ASKED QUESTIONS

6.1 My model reports cuda runtime error(2): out of memory

As the error message suggests, you have run out of memory on your GPU. Since we often deal with large amounts of data in PyTorch, small mistakes can rapidly cause your program to use up all of your GPU; fortunately, the fixes in these cases are often simple. Here are a few common things to check:

Dont accumulate history across your training loop. By default, computations involving variables that require gradients will keep history. This means that you should avoid using such variables in computations which will live beyond your training loops, e.g., when tracking statistics. Instead, you should detach the variable or access its underlying data.

Sometimes, it can be non-obvious when differentiable variables can occur. Consider the following training loop (abridged from source):

```
total_loss = 0
for i in range(10000):
    optimizer.zero_grad()
    output = model(input)
    loss = criterion(output)
    loss.backward()
    optimizer.step()
    total_loss += loss
```

Here, `total_loss` is accumulating history across your training loop, since `loss` is a differentiable variable with autograd history. You can fix this by writing *total_loss* += *float(loss)* instead.

Other instances of this problem: 1.

Dont hold onto tensors and variables you dont need. If you assign a Tensor or Variable to a local, Python will not deallocate until the local goes out of scope. You can free this reference by using `del x`. Similarly, if you assign a Tensor or Variable to a member variable of an object, it will not deallocate until the object goes out of scope. You will get the best memory usage if you dont hold onto temporaries you dont need.

The scopes of locals can be larger than you expect. For example:

```
for i in range(5):
    intermediate = f(input[i])
    result += g(intermediate)
output = h(result)
return output
```

Here, `intermediate` remains live even while `h` is executing, because its scope extrudes past the end of the loop. To free it earlier, you should `del intermediate` when you are done with it.

Dont run RNNs on sequences that are too large. The amount of memory required to backpropagate through an RNN scales linearly with the length of the RNN input; thus, you will run out of memory if you try to feed an RNN a sequence that is too long.

The technical term for this phenomenon is backpropagation through time, and there are plenty of references for how to implement truncated BPTT, including in the word language model example; truncation is handled by the `repackage` function as described in this forum post.

Dont use linear layers that are too large. A linear layer `nn.Linear(m, n)` uses $O(nm)$ memory: that is to say, the memory requirements of the weights scales quadratically with the number of features. It is very easy to blow through your memory this way (and remember that you will need at least twice the size of the weights, since you also need to store the gradients.)

6.2 My GPU memory isnt freed properly

PyTorch uses a caching memory allocator to speed up memory allocations. As a result, the values shown in `nvidia-smi` usually dont reflect the true memory usage. See *Memory management* for more details about GPU memory management.

If your GPU memory isnt freed even after Python quits, it is very likely that some Python subprocesses are still alive. You may find them via `ps -elf | grep python` and manually kill them with `kill -9 [pid]`.

6.3 My data loader workers return identical random numbers

You are likely using other libraries to generate random numbers in the dataset. For example, NumPys RNG is duplicated when worker subprocesses are started via `fork`. See *`torch.utils.data.DataLoader`*s documentation for how to properly set up random seeds in workers with its `worker_init_fn` option.

6.4 My recurrent network doesnt work with data parallelism

There is a subtlety in using the `pack sequence -> recurrent network -> unpack sequence` pattern in a *`Module`* with *`DataParallel`* or *`data_parallel()`*. Input to each the `forward()` on each device will only be part of the entire input. Because the unpack operation *`torch.nn.utils.rnn.pad_packed_sequence()`* by default only pads up to the longest input it sees, i.e., the longest on that particular device, size mismatches will happen when results are gathered together. Therefore, you can instead take advantage of the `total_length` argument of *`pad_packed_sequence()`* to make sure that the `forward()` calls return sequences of same length. For example, you can write:

```
from torch.nn.utils.rnn import pack_padded_sequence, pad_packed_sequence

class MyModule(nn.Module):
    # ... __init__, other methods, etc.

    # padded_input is of shape [B x T x *] (batch_first mode) and contains
    # the sequences sorted by lengths
    #   B is the batch size
    #   T is max sequence length
    def forward(self, padded_input, input_lengths):
        total_length = padded_input.size(1)  # get the max sequence length
        packed_input = pack_padded_sequence(padded_input, input_lengths,
                                            batch_first=True)
```

(continues on next page)

(continued from previous page)

```
        packed_output, _ = self.my_lstm(packed_input)
        output, _ = pad_packed_sequence(packed_output, batch_first=True,
                                        total_length=total_length)
        return output


m = MyModule().cuda()
dp_m = nn.DataParallel(m)
```

Additionally, extra care needs to be taken when batch dimension is dim `1` (i.e., `batch_first=False`) with data parallelism. In this case, the first argument of pack_padded_sequence `padding_input` will be of shape `[T x B x *]` and should be scattered along dim `1`, but the second argument `input_lengths` will be of shape `[B]` and should be scattered along dim `0`. Extra code to manipulate the tensor shapes will be needed.

CHAPTER

SEVEN

FEATURES FOR LARGE-SCALE DEPLOYMENTS

This note talks about several extension points and tricks that might be useful when running PyTorch within a larger system or operating multiple systems using PyTorch in a larger organization.

It doesnt cover topics of deploying models to production. Check `torch.jit` or one of the corresponding tutorials.

The note assumes that you either build PyTorch from source in your organization or have an ability to statically link additional code to be loaded when PyTorch is used. Therefore, many of the hooks are exposed as C++ APIs that can be triggered once in a centralized place, e.g. in static initialization code.

7.1 Fleet-wide operator profiling

PyTorch comes with `torch.autograd.profiler` capable of measuring time taken by individual operators on demand. One can use the same mechanism to do always ON measurements for any process running PyTorch. It might be useful for gathering information about PyTorch workloads running in a given process or across the entire set of machines.

New callbacks for any operator invocation can be added with `torch::autograd::profiler::pushCallback`. Hooks will be called with `torch::autograd::profiler::RecordFunction` struct that describes invocation context (e.g. *name*). If enabled, `RecordFunction::inputs()` contains arguments of the function represented as `torch::IValue` variant type. Note, that inputs logging is relatively expensive and thus has to be enabled explicitly.

The operator callbacks also have access to `at::getThreadLocalDebugInfo()` interface that returns a pointer to the struct holding the debug information. This debug information is supposed to be set earlier with the corresponding `at::setThreadLocalDebugInfo(debug_info)` call. Debug information is propagated through the forward (including async `fork` tasks) and backward passes and can be useful for passing some extra information about execution environment (e.g. model id) from the higher layers of the application down to the operator callbacks.

Invoking callbacks adds some overhead, so usually its useful to just randomly sample operator invocations. This can be enabled on per-callback basis with a global sampling rate specified by *torch::autograd::profiler::setSamplingProbability*.

Note, that `pushCallback` and `setSamplingProbability` are not thread-safe and can be called only when no PyTorch operator is running. Usually, its a good idea to call them once during initialization.

Heres an example:

```
// Called somewhere in the program beginning
void init() {
    // Sample one in a hundred operator runs randomly
    torch::autograd::setSamplingProbability(0.01);
    pushCallback(
        &onFunctionEnter,
        &onFunctionExit,
        /* needs_inputs */ true,
        /* sampled */ true
    );
}

void onFunctionEnter(const RecordFunction& fn) {
    std::cerr << "Before function " << fn.name()
              << " with " << fn.inputs().size() << " inputs" << std::endl;
}

void onFunctionExit(const RecordFunction& fn) {
    std::cerr << "After function " << fn.name();
}
```

7.2 API usage logging

When running in a broader ecosystem, for example in managed job scheduler, its often useful to track which binaries invoke particular PyTorch APIs. There exists simple instrumentation injected at several important API points that triggers a given callback. Because usually PyTorch is invoked in one-off python scripts, the callback fires only once for a given process for each of the APIs.

`c10::SetAPIUsageHandler` can be used to register API usage instrumentation handler. Passed argument is going to be an api key identifying used point, for example `python.import` for PyTorch extension import or `torch.script.compile` if TorchScript compilation was triggered.

```
SetAPIUsageLogger([](const std::string& event_name) {
    std::cerr << "API was used: " << event_name << std::endl;
});
```

Note for developers: new API trigger points can be added in code with `C10_LOG_API_USAGE_ONCE("my_api")` in C++ or `torch._C._log_api_usage_once("my.api")` in Python.

7.3 Attaching metadata to saved TorchScript models

TorchScript modules can be saved as an archive file that bundles serialized parameters and module code as TorchScript (see *`torch.jit.save()`*). Its often convenient to bundle additional information together with the model, for example, description of model producer or auxiliary artifacts.

It can be achieved by passing the `_extra_files` argument to *`torch.jit.save()`* and `torch::jit::load` to store and retrieve arbitrary binary blobs during saving process. Since TorchScript files are regular ZIP archives, extra information gets stored as regular files inside archives `extra/` directory.

Theres also a global hook allowing to attach extra files to any TorchScript archive produced in the current process. It might be useful to tag models with producer metadata, akin to JPEG metadata produced by digital cameras. Example usage might look like:

```
SetExportModuleExtraFilesHook([](const script::Module&) {
    script::ExtraFilesMap files;
    files["producer_info.json"] = "{\"user\": \"" + getenv("USER") + "\"}";
    return files;
});
```

7.4 Build environment considerations

TorchScripts compilation needs to have access to the original python files as it uses pythons `inspect.getsource` call. In certain production environments it might require explicitly deploying `.py` files along with precompiled `.pyc`.

7.5 Common extension points

PyTorch APIs are generally loosely coupled and its easy to replace a component with specialized version. Common extension points include:

- Custom operators implemented in C++ - see tutorial for more details.
- Custom data reading can be often integrated directly by invoking corresponding python library. Existing functionality of *`torch.utils.data`* can be utilized by extending *`Dataset`* or `IterableDataset`.

CHAPTER

EIGHT

MULTIPROCESSING BEST PRACTICES

`torch.multiprocessing` is a drop in replacement for Pythons `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.

Note: When a *Tensor* is sent to another process, the *Tensor* data is shared. If *torch.Tensor.grad* is not `None`, it is also shared. After a *Tensor* without a *torch.Tensor.grad* field is sent to the other process, it creates a standard process-specific `.grad` *Tensor* that is not automatically shared across all processes, unlike how the *Tensor*s data has been shared.

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

8.1 CUDA in multiprocessing

The CUDA runtime does not support the `fork` start method. However, `multiprocessing` in Python 2 can only create subprocesses using `fork`. So Python 3 and either `spawn` or `forkserver` start method are required to use CUDA in subprocesses.

Note: The start method can be set via either creating a context with `multiprocessing.get_context(...)` or directly using `multiprocessing.set_start_method(...)`.

Unlike CPU tensors, the sending process is required to keep the original tensor as long as the receiving process retains a copy of the tensor. It is implemented under the hood but requires users to follow the best practices for the program to run correctly. For example, the sending process must stay alive as long as the consumer process has references to the tensor, and the refcounting can not save you if the consumer process exits abnormally via a fatal signal. See this section.

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

8.2 Best practices and tips

8.2.1 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 theres any thread that holds a lock or imports a module, and `fork` is called, its 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

dont, 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 doesnt use any additional threads.

Were trying our best to make it easy for you and ensure these deadlocks dont happen but some things are out of our control. If you have any issues you cant cope with for a while, try reaching out on forums, and well see if its an issue we can fix.

8.2.2 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 its 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.

8.2.3 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 theyre sometimes a less elegant solution, will work properly in all cases.

> **Warning:** 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, theres 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):
```

(continues on next page)

(continued from previous page)

```
        p = mp.Process(target=train, args=(model,))
        p.start()
        processes.append(p)
    for p in processes:
        p.join()
```

CHAPTER

NINE

REPRODUCIBILITY

Completely reproducible results are not guaranteed across PyTorch releases, individual commits or different platforms. Furthermore, results need not be reproducible between CPU and GPU executions, even when using identical seeds.

However, in order to make computations deterministic on your specific problem on one specific platform and PyTorch release, there are a couple of steps to take.

There are two pseudorandom number generators involved in PyTorch, which you will need to seed manually to make runs reproducible. Furthermore, you should ensure that all other libraries your code relies on and which use random numbers also use a fixed seed.

9.1 PyTorch

You can use *torch.manual_seed()* to seed the RNG for all devices (both CPU and CUDA):

```
import torch
torch.manual_seed(0)
```

There are some PyTorch functions that use CUDA functions that can be a source of non-determinism. One class of such CUDA functions are atomic operations, in particular `atomicAdd`, where the order of parallel additions to the same value is undetermined and, for floating-point variables, a source of variance in the result. PyTorch functions that use `atomicAdd` in the forward include *torch.Tensor.index_add_()*, *torch.Tensor.scatter_add_()*, *torch.bincount()*.

A number of operations have backwards that use `atomicAdd`, in particular *torch.nn.functional.embedding_bag()*, *torch.nn.functional.ctc_loss()* and many forms of pooling, padding, and sampling. There currently is no simple way of avoiding non-determinism in these functions.

9.2 CuDNN

When running on the CuDNN backend, two further options must be set:

```
torch.backends.cudnn.deterministic = True
torch.backends.cudnn.benchmark = False
```

Warning: Deterministic mode can have a performance impact, depending on your model. This means that due to the deterministic nature of the model, the processing speed (i.e. processed batch items per second) can be lower than when the model is non-deterministic.

9.3 Numpy

If you or any of the libraries you are using rely on Numpy, you should seed the Numpy RNG as well. This can be done with:

```
import numpy as np
np.random.seed(0)
```

CHAPTER

TEN

SERIALIZATION SEMANTICS

10.1 Best practices

10.1.1 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

ELEVEN

WINDOWS FAQ

11.1 Building from source

11.1.1 Include optional components

There are two supported components for Windows PyTorch: MKL and MAGMA. Here are the steps to build with them.

```
REM Make sure you have 7z and curl installed.

REM Download MKL files
curl https://s3.amazonaws.com/ossci-windows/mkl_2018.2.185.7z -k -O
7z x -aoa mkl_2018.2.185.7z -omkl

REM Download MAGMA files
REM cuda100/cuda101 is also available for `CUDA_PREFIX`. There are also 2.4.0␣
↪binaries for cuda80/cuda92.
REM The configuration could be `debug` or `release` for 2.5.0. Only `release` is␣
↪available for 2.4.0.
set CUDA_PREFIX=cuda90
set CONFIG=release
curl -k https://s3.amazonaws.com/ossci-windows/magma_2.5.0_%CUDA_PREFIX%_%CONFIG%.7z -
↪o magma.7z
7z x -aoa magma.7z -omagma

REM Setting essential environment variables
set "CMAKE_INCLUDE_PATH=%cd%\\mkl\\include"
set "LIB=%cd%\\mkl\\lib;%LIB%"
set "MAGMA_HOME=%cd%\\magma"
```

11.1.2 Speeding CUDA build for Windows

Visual Studio doesnt support parallel custom task currently. As an alternative, we can use `Ninja` to parallelize CUDA build tasks. It can be used by typing only a few lines of code.

```
REM Let's install ninja first.
pip install ninja

REM Set it as the cmake generator
set CMAKE_GENERATOR=Ninja
```

11.1.3 One key install script

You can take a look at this set of scripts. It will lead the way for you.

11.2 Extension

11.2.1 CFFI Extension

The support for CFFI Extension is very experimental. Therere generally two steps to enable it under Windows.

First, specify additional `libraries` in `Extension` object to make it build on Windows.

```
ffi = create_extension(
    '_ext.my_lib',
    headers=headers,
    sources=sources,
    define_macros=defines,
    relative_to=__file__,
    with_cuda=with_cuda,
    extra_compile_args=["-std=c99"],
    libraries=['ATen', '_C'] # Append cuda libaries when necessary, like cudart
)
```

Second, here is a workground for unresolved external symbol state caused by `extern THCState *state;`

Change the source code from C to C++. An example is listed below.

```
#include <THC/THC.h>
#include <ATen/ATen.h>

THCState *state = at::globalContext().thc_state;

extern "C" int my_lib_add_forward_cuda(THCudaTensor *input1, THCudaTensor *input2,
                                       THCudaTensor *output)
{
    if (!THCudaTensor_isSameSizeAs(state, input1, input2))
    return 0;
    THCudaTensor_resizeAs(state, output, input1);
    THCudaTensor_cadd(state, output, input1, 1.0, input2);
    return 1;
}

extern "C" int my_lib_add_backward_cuda(THCudaTensor *grad_output, THCudaTensor *grad_
↪input)
{
    THCudaTensor_resizeAs(state, grad_input, grad_output);
    THCudaTensor_fill(state, grad_input, 1);
    return 1;
}
```

11.2.2 Cpp Extension

This type of extension has better support compared with the previous one. However, it still needs some manual configuration. First, you should open the **x86_x64 Cross Tools Command Prompt for VS 2017**. And then, you can start your compiling process.

11.3 Installation

11.3.1 Package not found in win-32 channel.

```
Solving environment: failed

PackagesNotFoundError: The following packages are not available from current channels:

- pytorch

Current channels:
- https://conda.anaconda.org/pytorch/win-32
- https://conda.anaconda.org/pytorch/noarch
- https://repo.continuum.io/pkgs/main/win-32
- https://repo.continuum.io/pkgs/main/noarch
- https://repo.continuum.io/pkgs/free/win-32
- https://repo.continuum.io/pkgs/free/noarch
- https://repo.continuum.io/pkgs/r/win-32
- https://repo.continuum.io/pkgs/r/noarch
- https://repo.continuum.io/pkgs/pro/win-32
- https://repo.continuum.io/pkgs/pro/noarch
- https://repo.continuum.io/pkgs/msys2/win-32
- https://repo.continuum.io/pkgs/msys2/noarch
```

PyTorch doesnt work on 32-bit system. Please use Windows and Python 64-bit version.

11.3.2 Why are there no Python 2 packages for Windows?

Because its not stable enough. Therere some issues that need to be solved before we officially release it. You can build it by yourself.

11.3.3 Import error

```
from torch._C import *

ImportError: DLL load failed: The specified module could not be found.
```

The problem is caused by the missing of the essential files. Actually, we include almost all the essential files that PyTorch need for the conda package except VC2017 redistributable and some mkl libraries. You can resolve this by typing the following command.

```
conda install -c peterjc123 vc vs2017_runtime
conda install mkl_fft intel_openmp numpy mkl
```

As for the wheels package, since we didnt pack some libaries and VS2017 redistributable files in, please make sure you install them manually. The VS 2017 redistributable installer can be downloaded. And you should also pay attention to your installation of Numpy. Make sure it uses MKL instead of OpenBLAS. You may type in the following command.

```
pip install numpy mkl intel-openmp mkl_fft
```

Another possible cause may be you are using GPU version without NVIDIA graphics cards. Please replace your GPU package with the CPU one.

```
from torch._C import *

ImportError: DLL load failed: The operating system cannot run %1.
```

This is actually an upstream issue of Anaconda. When you initialize your environment with conda-forge channel, this issue will emerge. You may fix the intel-openmp libraries through this command.

```
conda install -c defaults intel-openmp -f
```

11.4 Usage (multiprocessing)

11.4.1 Multiprocessing error without if-clause protection

```
RuntimeError:
       An attempt has been made to start a new process before the
       current process has finished its bootstrapping phase.

   This probably means that you are not using fork to start your
   child processes and you have forgotten to use the proper idiom
   in the main module:

       if __name__ == '__main__':
           freeze_support()
           ...

   The "freeze_support()" line can be omitted if the program
   is not going to be frozen to produce an executable.
```

The implementation of `multiprocessing` is different on Windows, which uses `spawn` instead of `fork`. So we have to wrap the code with an if-clause to protect the code from executing multiple times. Refactor your code into the following structure.

```
import torch

def main()
    for i, data in enumerate(dataloader):
        # do something here

if __name__ == '__main__':
    main()
```

11.4.2 Multiprocessing error Broken pipe

```
ForkingPickler(file, protocol).dump(obj)

BrokenPipeError: [Errno 32] Broken pipe
```

This issue happens when the child process ends before the parent process finishes sending data. There may be something wrong with your code. You can debug your code by reducing the `num_worker` of *`DataLoader`* to zero and see if the issue persists.

11.4.3 Multiprocessing error driver shut down

```
Couldnt open shared file mapping: <torch_14808_1591070686>, error code: <1455> at␣
↪torch\lib\TH\THAllocator.c:154

[windows] driver shut down
```

Please update your graphics driver. If this persists, this may be that your graphics card is too old or the calculation is too heavy for your card. Please update the TDR settings according to this post.

11.4.4 CUDA IPC operations

```
THCudaCheck FAIL file=torch\csrc\generic\StorageSharing.cpp line=252 error=63 : OS␣
↪call failed or operation not supported on this OS
```

They are not supported on Windows. Something like doing multiprocessing on CUDA tensors cannot succeed, there are two alternatives for this.

1. Dont use `multiprocessing`. Set the `num_worker` of *`DataLoader`* to zero.

2. Share CPU tensors instead. Make sure your custom `DataSet` returns CPU tensors.

CHAPTER

TWELVE

PYTORCH CONTRIBUTION GUIDE

PyTorch is a GPU-accelerated Python tensor computation package for building deep neural networks built on tape-based autograd systems.

12.1 The PyTorch Contribution Process

The PyTorch organization is governed by PyTorch Governance.

The PyTorch development process involves a healthy amount of open discussions between the core development team and the community.

PyTorch operates similar to most open source projects on GitHub. However, if youve never contributed to an open source project before, here is the basic process.

- **Figure out what youre going to work on.** The majority of open source contributions come from people scratching their own itches. However, if you dont know what you want to work on, or are just looking to get more acquainted with the project, here are some tips for how to find appropriate tasks:
  - Look through the issue tracker and see if there are any issues you know how to fix. Issues that are confirmed by other contributors tend to be better to investigate. We also maintain some labels for issues which are likely to be good for new people, e.g., **bootcamp** and **1hr**, although these labels are less well maintained.
  - Join us on Slack and let us know youre interested in getting to know PyTorch. Were very happy to help out researchers and partners get up to speed with the codebase.
- **Figure out the scope of your change and reach out for design comments on a GitHub issue if its large.** The majority of pull requests are small; in that case, no need to let us know about what you want to do, just get cracking. But if the change is going to be large, its usually a good idea to get some design comments about it first.
  - If you dont know how big a change is going to be, we can help you figure it out! Just post about it on issues or Slack.
  - Some feature additions are very standardized; for example, lots of people add new operators or optimizers to PyTorch. Design discussion in these cases boils down mostly to, Do we want this operator/optimizer? Giving evidence for its utility, e.g., usage in peer reviewed papers, or existence in other frameworks, helps a bit when making this case.
    - **Adding operators / algorithms from recently-released research** is generally not accepted, unless there is overwhelming evidence that this newly published work has ground-breaking results and will eventually become a standard in the field. If you are not sure where your method falls, open an issue first before implementing a PR.

  - Core changes and refactors can be quite difficult to coordinate, as the pace of development on PyTorch master is quite fast. Definitely reach out about fundamental or cross-cutting changes; we can often give guidance about how to stage such changes into more easily reviewable pieces.

- **Code it out!**
  - See the technical guide for advice for working with PyTorch in a technical form.
- **Open a pull request.**
  - If you are not ready for the pull request to be reviewed, tag it with [WIP]. We will ignore it when doing review passes. If you are working on a complex change, its good to start things off as WIP, because you will need to spend time looking at CI results to see if things worked out or not.
  - Find an appropriate reviewer for your change. We have some folks who regularly go through the PR queue and try to review everything, but if you happen to know who the maintainer for a given subsystem affected by your patch is, feel free to include them directly on the pull request. You can learn more about this structure at PyTorch Subsystem Ownership.
- **Iterate on the pull request until its accepted!**
  - Well try our best to minimize the number of review roundtrips and block PRs only when there are major issues. For the most common issues in pull requests, take a look at *Common Mistakes*.
  - Once a pull request is accepted and CI is passing, there is nothing else you need to do; we will merge the PR for you.

12.2 Getting Started

12.2.1 Proposing new features

New feature ideas are best discussed on a specific issue. Please include as much information as you can, any accompanying data, and your proposed solution. The PyTorch team and community frequently reviews new issues and comments where they think they can help. If you feel confident in your solution, go ahead and implement it.

12.2.2 Reporting Issues

If youve identified an issue, first search through the list of existing issues on the repo. If you are unable to find a similar issue, then create a new one. Supply as much information you can to reproduce the problematic behavior. Also, include any additional insights like the behavior you expect.

12.2.3 Implementing Features or Fixing Bugs

If you want to fix a specific issue, its best to comment on the individual issue with your intent. However, we do not lock or assign issues except in cases where we have worked with the developer before. Its best to strike up a conversation on the issue and discuss your proposed solution. The PyTorch team can provide guidance that saves you time.

Issues that are labeled first-new-issue, low, or medium priority provide the best entrance point are great places to start.

12.2.4 Adding Tutorials

A great deal of the tutorials on pytorch.org come from the community itself and we welcome additional contributions. To learn more about how to contribute a new tutorial you can learn more here: PyTorch.org Tutorial Contribution Guide on Github

12.2.5 Improving Documentation & Tutorials

We aim to produce high quality documentation and tutorials. On rare occasions that content includes typos or bugs. If you find something you can fix, send us a pull request for consideration.

Take a look at the *Documentation* section to learn how our system works.

12.2.6 Participating in online discussions

You can find active discussions happening on the PyTorch Discussion forum.

12.2.7 Submitting pull requests to fix open issues

You can view a list of all open issues here. Commenting on an issue is a great way to get the attention of the team. From here you can share your ideas and how you plan to resolve the issue.

For more challenging issues, the team will provide feedback and direction for how to best solve the issue.

If youre not able to fix the issue itself, commenting and sharing whether you can reproduce the issue can be useful for helping the team identify problem areas.

12.2.8 Reviewing open pull requests

We appreciate your help reviewing and commenting on pull requests. Our team strives to keep the number of open pull requests at a manageable size, we respond quickly for more information if we need it, and we merge PRs that we think are useful. However, due to the high level of interest, additional eyes on pull requests is appreciated.

12.2.9 Improving code readability

Improve code readability helps everyone. It is often better to submit a small number of pull requests that touch few files versus a large pull request that touches many files. Starting a discussion in the PyTorch forum here or on an issue related to your improvement is the best way to get started.

12.2.10 Adding test cases to make the codebase more robust

Additional test coverage is appreciated.

12.2.11 Promoting PyTorch

Your use of PyTorch in your projects, research papers, write ups, blogs, or general discussions around the internet helps to raise awareness for PyTorch and our growing community. Please reach out to pytorch-marketing@fb.com for marketing support.

12.2.12 Triaging issues

If you feel that an issue could benefit from a particular tag or level of complexity comment on the issue and share your opinion. If an you feel an issue isnt categorized properly comment and let the team know.

12.3 About open source development

If this is your first time contributing to an open source project, some aspects of the development process may seem unusual to you.

- **There is no way to claim issues.** People often want to claim an issue when they decide to work on it, to ensure that there isnt wasted work when someone else ends up working on it. This doesnt really work too well in open source, since someone may decide to work on something, and end up not having time to do it. Feel free to give information in an advisory fashion, but at the end of the day, we will take running code and rough consensus.
- **There is a high bar for new functionality that is added.** Unlike in a corporate environment, where the person who wrote code implicitly owns it and can be expected to take care of it in the beginning of its lifetime, once a pull request is merged into an open source project, it immediately becomes the collective responsibility of all maintainers on the project. When we merge code, we are saying that we, the maintainers, are able to review subsequent changes and make a bugfix to the code. This naturally leads to a higher standard of contribution.

12.4 Common Mistakes To Avoid

- **Did you add tests?** (Or if the change is hard to test, did you describe how you tested your change?)
  - We have a few motivations for why we ask for tests:
    1. to help us tell if we break it later
    2. to help us tell if the patch is correct in the first place (yes, we did review it, but as Knuth says, beware of the following code, for I have not run it, merely proven it correct)
  - When is it OK not to add a test? Sometimes a change cant be conveniently tested, or the change is so obviously correct (and unlikely to be broken) that its OK not to test it. On the contrary, if a change is seems likely (or is known to be likely) to be accidentally broken, its important to put in the time to work out a testing strategy.
- **Is your PR too long?**
  - Its easier for us to review and merge small PRs. Difficulty of reviewing a PR scales nonlinearly with its size.
  - When is it OK to submit a large PR? It helps a lot if there was a corresponding design discussion in an issue, with sign off from the people who are going to review your diff. We can also help give advice about how to split up a large change into individually shippable parts. Similarly, it helps if there is a complete description of the contents of the PR: its easier to review code if we know whats inside!
- **Comments for subtle things?** In cases where behavior of your code is nuanced, please include extra comments and documentation to allow us to better understand the intention of your code.
- **Did you add a hack?** Sometimes a hack is the right answer. But usually we will have to discuss it.
- **Do you want to touch a very core component?** In order to prevent major regressions, pull requests that touch core components receive extra scrutiny. Make sure youve discussed your changes with the team before undertaking major changes.
- **Want to add a new feature?** If you want to add new features, comment your intention on the related issue. Our team tries to comment on and provide feedback to the community. Its better to have an open discussion with the team and the rest of the community prior to building new features. This helps us stay aware of what youre working on and increases the chance that itll be merged.
- **Did you touch unrelated code to the PR?** To aid in code review, please only include files in your pull request that are directly related to your changes.

Frequently asked questions

- **How can I contribute as a reviewer?** There is lots of value if community developer reproduce issues, try out new functionality, or otherwise help us identify or troubleshoot issues. Commenting on tasks or pull requests with your enviroment details is helpful and appreciated.
- **CI tests failed, what does it mean?** Maybe you need to merge with master or rebase with latest changes. Pushing your changes should re-trigger CI tests. If the tests persist, youll want to trace through the error messages and resolve the related issues.
- **What are the most high risk changes?** Anything that touches build configuration is an risky area. Please avoid changing these unless youve had a discussion with the team beforehand.
- **Hey, a commit showed up on my branch, whats up with that?** Sometimes another community member will provide a patch or fix to your pull request or branch. This is often needed for getting CI tests to pass.

12.5 On Documentation

12.5.1 Python Docs

PyTorch documentation is generated from python source using Sphinx. Generated HTML is copied to the docs folder in the master branch of pytorch.github.io, and is served via GitHub pages.

- Site: http://pytorch.org/docs
- GitHub: https://github.com/pytorch/pytorch/tree/master/docs
- Served from: https://github.com/pytorch/pytorch.github.io/tree/master/doc

12.5.2 C++ Docs

For C++ code we use Doxygen to generate the content files. The C++ docs are built on a special server and the resulting files are copied to the https://github.com/pytorch/cppdocs repo, and are served from GitHub pages.

- Site: http://pytorch.org/cppdocs
- GitHub: https://github.com/pytorch/pytorch/tree/master/docs/cpp
- Served from: https://github.com/pytorch/cppdocs

12.6 Tutorials

PyTorch tutorials are documents used to help understand using PyTorch to accomplish specific tasks or to understand more holistic concepts. Tutorials are built using Sphinx-Gallery from executable python sources files, or from restructured-text (rst) files.

- Site: http://pytorch.org/tutorials
- GitHub: http://github.com/pytorch/tutorials

12.6.1 Tutorials Build Overview

For tutorials, pull requests trigger a rebuild the entire site using CircleCI to test the effects of the change. This build is sharded into 9 worker builds and takes around 40 minutes total. At the same time, we do a Netlify build using *make html-noplot*, which builds the site without rendering the notebook output into pages for quick review.

After a PR is accepted, the site is rebuilt and deployed from CircleCI.

12.6.2 Contributing a new Tutorial

PyTorch.org Tutorial Contribution Guide

CHAPTER

THIRTEEN

PYTORCH GOVERNANCE

13.1 Governance Philosophy and Guiding Tenets

PyTorch adopts a governance structure with a small set of maintainers driving the overall project direction with a strong bias towards PyTorchs design philosophy where design and code contributions are valued. Beyond the core maintainers, there is also a slightly broader set of core developers that have the ability to directly merge pull requests and own various parts of the core code base.

Beyond the maintainers and core devs, the community is encouraged to contribute, file issues, make proposals, review pull requests and be present in the community. Given contributions and willingness to invest, anyone can be provided write access or ownership of parts of the codebase.

Based on this governance structure, the project has the following core operating tenets by which decisions are made and overall culture is derived:

1. **Code contributions** matter much more than corporate sponsorship and independent developers are highly valued.
2. **Project influence** is gained through contributions (whether PRs, forum answers, code reviews or otherwise)

13.2 Key people and their functions

13.2.1 Project Maintainers

Project maintainers provide leadership and direction for the PyTorch project. Specifics include:

- Articulate a cohesive long-term vision for the project
- Possess a deep understanding of the PyTorch code base
- Negotiate and resolve contentious issues in ways acceptable to all parties involved

PyTorch Maintainers:

- Adam Paszke (apaszke)
- Soumith Chintala (soumith)
- Edward Yang (ezyang)
- Greg Chanan (gchanan)
- Dmytro Dzhulgakov (dzhulgakov)
- (sunsetting) Sam Gross (colesbury)

13.2.2 Core Developers

The PyTorch project is developed by a team of core developers. You can find the list of core developers at PyTorch Governance | Persons of Interest.

While membership is determined by presence in the PyTorch core team in the PyTorch organization on GitHub, contribution takes many forms:

- committing changes to the repository;
- reviewing pull requests by others;
- triaging bug reports on the issue tracker;
- discussing topics on official PyTorch communication channels.

13.2.3 Moderators

There is a group of people, some of which are not core developers, responsible for ensuring that discussions on official communication channels adhere to the Code of Conduct. They take action in view of violations and help to support a healthy community. You can find the list of moderators here.

13.3 Decision Making

13.3.1 Uncontroversial Changes

Primary work happens through bug tracker issues and pull requests on GitHub. Core developers should avoid pushing their changes directly to the PyTorch repository, instead relying on pull requests. Approving a pull request by a core developer allows it to be merged without further process. Core Developers and Project Maintainers ultimately approve these changes.

Notifying relevant experts about a bug tracker issue or a pull request is important. Reviews from experts in the given interest area are strongly preferred, especially on pull request approvals. Failure to do so might end up with the change being reverted by the relevant expert.

13.3.2 Controversial decision process

Substantial changes in a given interest area require a GitHub issue to be opened for discussion. This includes:

- Any semantic or syntactic change to the framework.
- Backwards-incompatible changes to the Python or Cpp API.
- Additions to the core framework, including substantial new functionality within an existing library.
- Removing core features

Project Maintainers ultimately approve these changes.

13.4 FAQ

Q: What if I would like to own (or partly own) a part of the project such as a domain api (i.e. Torch Vision)? This is absolutely possible. The first step is to start contributing to the existing project area and contributing to its

health and success. In addition to this, you can make a proposal through a GitHub issue for new functionality or changes to improve the project area.

Q: What if I am a company looking to use PyTorch internally for development, can I be granted or purchase a board seat to drive the project direction? No, the PyTorch project is strictly driven by the maintainer-driven project philosophy and does not have a board or vehicle to take financial contributions relating to gaining influence over technical direction.

Q: Does the PyTorch project support grants or ways to support independent developers using or contributing to the project? No, not at this point. We are however looking at ways to better support the community of independent developers around PyTorch. If you have suggestions or inputs, please reach out on the PyTorch forums to discuss.

Q: How do I contribute code to the project? If the change is relatively minor, a pull request on GitHub can be opened up immediately for review and merge by the project committers. For larger changes, please open an issue to make a proposal to discuss prior. Please also see the PyTorch Contributor Guide for contribution guidelines.

Q: Can I become a committer on the project? Unfortunately, the current commit process to PyTorch involves an interaction with Facebook infrastructure that can only be triggered by Facebook employees. We are however looking at ways to expand the committer base to individuals outside of Facebook and will provide an update when the tooling exists to allow this.

Q: What if i would like to deliver a PyTorch tutorial at a conference or otherwise? Do I need to be officially a committer to do this? No, we encourage community members to showcase their work wherever and whenever they can. Please reach out to pytorch-marketing@fb.com for marketing support.

CHAPTER

FOURTEEN

PYTORCH GOVERNANCE | PERSONS OF INTEREST

14.1 General Maintainers

- Adam Paszke (apaszke)
- Soumith Chintala (soumith)
- Edward Yang (ezyang)
- Greg Chanan (gchanan)
- Dmytro Dzhulgakov (dzhulgakov)
- (sunsetting) Sam Gross (colesbury)

14.2 Module-level maintainers

14.2.1 torch.*

- Greg Chanan (gchanan)
- Soumith Chintala (soumith)
- [linear algebra] Vishwak Srinivasan (vishwakftw)

14.2.2 torch.nn

- Thomas Viehmann (t-vi)
- Adam Paszke (apaszke)
- Greg Chanan (gchanan)
- Soumith Chintala (soumith)
- Sam Gross (colesbury)

14.2.3 torch.optim

- Vincent Quenneville-Belair (vincentqb)
- Soumith Chintala (soumith)

14.2.4 Autograd Engine

- Edward Yang (ezyang)
- Alban Desmaison (alband)
- Adam Paszke (apaszke)

14.2.5 JIT

- Zach Devito (zdevito)
- Michael Suo (suo)

14.2.6 Distributions & RNG

- Fritz Obermeyer (fritzo)
- Neeraj Pradhan (neerajprad)
- Alican Bozkurt (alicanb)
- Vishwak Srinivasan (vishwakftw)

14.2.7 Distributed

- Pieter Noordhuis (pietern)
- Shen Li (mrshenli)
- (proposed) Pritam Damania (pritamdamania87)

14.2.8 Multiprocessing and DataLoaders

- Vitaly Fedyunin (VitalyFedyunin)
- Simon Wang (SsnL)
- Adam Paszke (apaszke)

14.2.9 CPU Performance / SIMD

- Xiaoqiang Zheng (zheng-xq)
- Vitaly Fedyunin (VitalyFedyunin)
- Sam Gross (colesbury)
- (sunsetting) Christian Puhrsch (cpuhrsch)
- [threading] Ilia Cherniavskii (ilia-cher)

14.2.10 CUDA

- Natalia Gimelshein (ngimel)
- Edward Yang (ezyang)
- Xiaoqiang Zheng (zheng-xq)

14.2.11 MKLDNN

- Junjie Bai (bddppq)
- Yinghai Lu (yinghai)

14.2.12 XLA

- Ailing Zhang (ailzhang)
- Gregory Chanan (gchanan)
- Davide Libenzi (dlibenzi)
- Alex Suhan (asuhan)

14.2.13 AMD/ROCm/HIP

- Junjie Bai (bddppq)
- Johannes M. Dieterich (iotamudelta)

14.2.14 Build + CI

- Will Feng (yf225)
- Edward Yang (ezyang)
- Soumith Chintala (soumith)
- Karl Ostmo (kostmo)
- Hong Xu (xuhdev)

14.2.15 Benchmarks

- Mingzhe Li (mingzhe09088)

14.2.16 C++ API

- Will Feng (yf225)

14.2.17 C10 utils and operator dispatch

- Sebastian Messmer (smessmer)
- Dmytro Dzhulgakov (dzhulgakov)

14.2.18 ONNX <-> PyTorch

- Lu Fang (houseroad)

14.2.19 Windows

- Peter Johnson (peterjc123)

14.2.20 PowerPC

- Alfredo Mendoza (avmgithub)

CHAPTER

FIFTEEN

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 >= 3.0.

15.1 Tensors

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

Parameters **obj** (*Object*) – Object to test

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

Parameters **obj** (*Object*) – Object to test

`torch.`**`is_floating_point`**(*tensor*) -> *(bool)*
Returns True if the data type of *tensor* is a floating point data type i.e., one of `torch.float64`, `torch.float32` and `torch.float16`.

Parameters **tensor** (*Tensor*) – the PyTorch tensor to test

`torch.`**`set_default_dtype`**(*d*)
Sets the default floating point dtype to `d`. This type will be used as default floating point type for type inference in *torch.tensor()*.

The default floating point dtype is initially `torch.float32`.

Parameters **d** (*torch.dtype*) – the floating point dtype to make the default

Example:

```
>>> torch.tensor([1.2, 3]).dtype           # initial default for floating point␣
↪is torch.float32
torch.float32
>>> torch.set_default_dtype(torch.float64)
>>> torch.tensor([1.2, 3]).dtype           # a new floating point tensor
torch.float64
```

`torch.`**`get_default_dtype`**() → torch.dtype
Get the current default floating point *torch.dtype*.

Example:

```
>>> torch.get_default_dtype()  # initial default for floating point is torch.
↪float32
torch.float32
>>> torch.set_default_dtype(torch.float64)
>>> torch.get_default_dtype()  # default is now changed to torch.float64
torch.float64
>>> torch.set_default_tensor_type(torch.FloatTensor)  # setting tensor type also␣
↪affects this
>>> torch.get_default_dtype()  # changed to torch.float32, the dtype for torch.
↪FloatTensor
torch.float32
```

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

Sets the default `torch.Tensor` type to floating point tensor type `t`. This type will also be used as default floating point type for type inference in *torch.tensor()*.

The default floating point tensor type is initially `torch.FloatTensor`.

Parameters **t** (*type or string*) – the floating point tensor type or its name

Example:

```
>>> torch.tensor([1.2, 3]).dtype    # initial default for floating point is torch.
↪float32
torch.float32
>>> torch.set_default_tensor_type(torch.DoubleTensor)
>>> torch.tensor([1.2, 3]).dtype    # a new floating point tensor
torch.float64
```

torch.**numel**(*input*) → int

Returns the total number of elements in the `input` tensor.

Parameters **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*, *sci_mode=None*)

Set options for printing. Items shamelessly taken from NumPy

Parameters

- **precision** – Number of digits of precision for floating point output (default = 4).
- **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 matrices will ignore this parameter.

- **profile** – Sane defaults for pretty printing. Can override with any of the above options. (any one of *default*, *short*, *full*)
- **sci_mode** – Enable (True) or disable (False) scientific notation. If None (default) is specified, the value is defined by *_Formatter*

torch.**set_flush_denormal**(*mode*) → bool

Disables denormal floating numbers on CPU.

Returns `True` if your system supports flushing denormal numbers and it successfully configures flush denormal mode. *set_flush_denormal()* is only supported on x86 architectures supporting SSE3.

Parameters **mode** (*bool*) – Controls whether to enable flush denormal mode or not

Example:

```
>>> torch.set_flush_denormal(True)
True
>>> torch.tensor([1e-323], dtype=torch.float64)
tensor([ 0.], dtype=torch.float64)
>>> torch.set_flush_denormal(False)
True
>>> torch.tensor([1e-323], dtype=torch.float64)
tensor(9.88131e-324 *
       [ 1.0000], dtype=torch.float64)
```

15.1.1 Creation Ops

Note: Random sampling creation ops are listed under *Random sampling* and include: *torch.rand()* *torch.rand_like()* *torch.randn()* *torch.randn_like()* *torch.randint()* *torch.randint_like()* *torch.randperm()* You may also use *torch.empty()* with the *In-place random sampling* methods to create *torch.Tensor* s with values sampled from a broader range of distributions.

torch.**tensor**(*data*, *dtype=None*, *device=None*, *requires_grad=False*, *pin_memory=False*) → Tensor

Constructs a tensor with `data`.

Warning: *torch.tensor()* always copies `data`. If you have a Tensor `data` and want to avoid a copy, use *torch.Tensor.requires_grad_()* or *torch.Tensor.detach()*. If you have a NumPy `ndarray` and want to avoid a copy, use *torch.as_tensor()*.

Warning: When data is a tensor *x*, *torch.tensor()* reads out the data from whatever it is passed, and constructs a leaf variable. Therefore `torch.tensor(x)` is equivalent to `x.clone().detach()` and `torch.tensor(x, requires_grad=True)` is equivalent to `x.clone().detach().requires_grad_(True)`. The equivalents using `clone()` and `detach()` are recommended.

Parameters

- **data** (*array_like*) – Initial data for the tensor. Can be a list, tuple, NumPy `ndarray`, scalar, and other types.

- **dtype** (*torch.dtype*, optional) – the desired data type of returned tensor. Default: if `None`, infers data type from `data`.
- **device** (*torch.device*, optional) – the desired device of returned tensor. Default: if `None`, uses the current device for the default tensor type (see *torch.set_default_tensor_type()*). `device` will be the CPU for CPU tensor types and the current CUDA device for CUDA tensor types.
- **requires_grad** (*bool, optional*) – If autograd should record operations on the returned tensor. Default: `False`.
- **pin_memory** (*bool, optional*) – If set, returned tensor would be allocated in the pinned memory. Works only for CPU tensors. Default: `False`.

Example:

```
>>> torch.tensor([[0.1, 1.2], [2.2, 3.1], [4.9, 5.2]])
tensor([[ 0.1000,  1.2000],
        [ 2.2000,  3.1000],
        [ 4.9000,  5.2000]])

>>> torch.tensor([0, 1])  # Type inference on data
tensor([ 0,  1])

>>> torch.tensor([[0.11111, 0.222222, 0.3333333]],
                 dtype=torch.float64,
                 device=torch.device('cuda:0'))  # creates a torch.cuda.
↪DoubleTensor
tensor([[ 0.1111,  0.2222,  0.3333]], dtype=torch.float64, device='cuda:0')

>>> torch.tensor(3.14159)  # Create a scalar (zero-dimensional tensor)
tensor(3.1416)

>>> torch.tensor([])  # Create an empty tensor (of size (0,))
tensor([])
```

torch.**sparse_coo_tensor**(*indices*, *values*, *size=None*, *dtype=None*, *device=None*, *requires_grad=False*) → Tensor

Constructs a sparse tensors in COO(rdinate) format with non-zero elements at the given `indices` with the given `values`. A sparse tensor can be *uncoalesced*, in that case, there are duplicate coordinates in the indices, and the value at that index is the sum of all duplicate value entries: torch.sparse.

Parameters

- **indices** (*array_like*) – Initial data for the tensor. Can be a list, tuple, NumPy `ndarray`, scalar, and other types. Will be cast to a `torch.LongTensor` internally. The indices are the coordinates of the non-zero values in the matrix, and thus should be two-dimensional where the first dimension is the number of tensor dimensions and the second dimension is the number of non-zero values.
- **values** (*array_like*) – Initial values for the tensor. Can be a list, tuple, NumPy `ndarray`, scalar, and other types.
- **size** (list, tuple, or `torch.Size`, optional) – Size of the sparse tensor. If not provided the size will be inferred as the minimum size big enough to hold all non-zero elements.
- **dtype** (*torch.dtype*, optional) – the desired data type of returned tensor. Default: if None, infers data type from `values`.

- **device** (*torch.device*, optional) – the desired device of returned tensor. Default: if None, uses the current device for the default tensor type (see *torch.set_default_tensor_type()*). `device` will be the CPU for CPU tensor types and the current CUDA device for CUDA tensor types.
- **requires_grad** (*bool, optional*) – If autograd should record operations on the returned tensor. Default: `False`.

Example:

```
>>> i = torch.tensor([[0, 1, 1],
                      [2, 0, 2]])
>>> v = torch.tensor([3, 4, 5], dtype=torch.float32)
>>> torch.sparse_coo_tensor(i, v, [2, 4])
tensor(indices=tensor([[0, 1, 1],
                       [2, 0, 2]]),
       values=tensor([3., 4., 5.]),
       size=(2, 4), nnz=3, layout=torch.sparse_coo)

>>> torch.sparse_coo_tensor(i, v)  # Shape inference
tensor(indices=tensor([[0, 1, 1],
                       [2, 0, 2]]),
       values=tensor([3., 4., 5.]),
       size=(2, 3), nnz=3, layout=torch.sparse_coo)

>>> torch.sparse_coo_tensor(i, v, [2, 4],
                            dtype=torch.float64,
                            device=torch.device('cuda:0'))
tensor(indices=tensor([[0, 1, 1],
                       [2, 0, 2]]),
       values=tensor([3., 4., 5.]),
       device='cuda:0', size=(2, 4), nnz=3, dtype=torch.float64,
       layout=torch.sparse_coo)

# Create an empty sparse tensor with the following invariants:
#   1. sparse_dim + dense_dim = len(SparseTensor.shape)
#   2. SparseTensor._indices().shape = (sparse_dim, nnz)
#   3. SparseTensor._values().shape = (nnz, SparseTensor.shape[sparse_dim:])
#
# For instance, to create an empty sparse tensor with nnz = 0, dense_dim = 0 and
# sparse_dim = 1 (hence indices is a 2D tensor of shape = (1, 0))
>>> S = torch.sparse_coo_tensor(torch.empty([1, 0]), [], [1])
tensor(indices=tensor([], size=(1, 0)),
       values=tensor([], size=(0,)),
       size=(1,), nnz=0, layout=torch.sparse_coo)

# and to create an empty sparse tensor with nnz = 0, dense_dim = 1 and
# sparse_dim = 1
>>> S = torch.sparse_coo_tensor(torch.empty([1, 0]), torch.empty([0, 2]), [1, 2])
tensor(indices=tensor([], size=(1, 0)),
       values=tensor([], size=(0, 2)),
       size=(1, 2), nnz=0, layout=torch.sparse_coo)
```

`torch.`**`as_tensor`**(*data*, *dtype=None*, *device=None*) → Tensor

Convert the data into a *torch.Tensor*. If the data is already a *Tensor* with the same *dtype* and *device*, no copy will be performed, otherwise a new *Tensor* will be returned with computational graph retained if data *Tensor* has `requires_grad=True`. Similarly, if the data is an `ndarray` of the corresponding *dtype* and the *device* is the cpu, no copy will be performed.

Parameters

- **data** (*array_like*) – Initial data for the tensor. Can be a list, tuple, NumPy `ndarray`, scalar, and other types.
- **dtype** (*torch.dtype*, optional) – the desired data type of returned tensor. Default: if `None`, infers data type from `data`.
- **device** (*torch.device*, optional) – the desired device of returned tensor. Default: if `None`, uses the current device for the default tensor type (see *torch.set_default_tensor_type()*). `device` will be the CPU for CPU tensor types and the current CUDA device for CUDA tensor types.

Example:

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

>>> a = numpy.array([1, 2, 3])
>>> t = torch.as_tensor(a, device=torch.device('cuda'))
>>> t
tensor([ 1,  2,  3])
>>> t[0] = -1
>>> a
array([1,  2,  3])
```

`torch.`**`as_strided`**`()`

`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
tensor([ 1,  2,  3])
>>> t[0] = -1
>>> a
array([-1,  2,  3])
```

`torch.`**`zeros`**(**sizes*, *out=None*, *dtype=None*, *layout=torch.strided*, *device=None*, *requires_grad=False*) → Tensor

Returns a tensor filled with the scalar value *0*, with the shape defined by the variable argument `sizes`.

Parameters

- **sizes** (*int...*) – a sequence of integers defining the shape of the output tensor. Can be a variable number of arguments or a collection like a list or tuple.
- **out** (*Tensor, optional*) – the output tensor
- **dtype** (*torch.dtype*, optional) – the desired data type of returned tensor. Default: if `None`, uses a global default (see *torch.set_default_tensor_type()*).

- **layout** (*torch.layout*, optional) – the desired layout of returned Tensor. Default: `torch.strided`.
- **device** (*torch.device*, optional) – the desired device of returned tensor. Default: if `None`, uses the current device for the default tensor type (see *torch.set_default_tensor_type()*). `device` will be the CPU for CPU tensor types and the current CUDA device for CUDA tensor types.
- **requires_grad** (*bool, optional*) – If autograd should record operations on the returned tensor. Default: `False`.

Example:

```
>>> torch.zeros(2, 3)
tensor([[ 0.,  0.,  0.],
        [ 0.,  0.,  0.]])

>>> torch.zeros(5)
tensor([ 0.,  0.,  0.,  0.,  0.])
```

`torch.`**`zeros_like`**(*input*, *dtype=None*, *layout=None*, *device=None*, *requires_grad=False*) → Tensor

Returns a tensor filled with the scalar value *0*, with the same size as `input`. `torch.zeros_like(input)` is equivalent to `torch.zeros(input.size(), dtype=input.dtype, layout=input.layout, device=input.device)`.

> **Warning:** As of 0.4, this function does not support an `out` keyword. As an alternative, the old `torch.zeros_like(input, out=output)` is equivalent to `torch.zeros(input.size(), out=output)`.

Parameters

- **input** (`Tensor`) – the size of `input` will determine size of the output tensor
- **dtype** (*torch.dtype*, optional) – the desired data type of returned Tensor. Default: if `None`, defaults to the dtype of `input`.
- **layout** (*torch.layout*, optional) – the desired layout of returned tensor. Default: if `None`, defaults to the layout of `input`.
- **device** (*torch.device*, optional) – the desired device of returned tensor. Default: if `None`, defaults to the device of `input`.
- **requires_grad** (*bool, optional*) – If autograd should record operations on the returned tensor. Default: `False`.

Example:

```
>>> input = torch.empty(2, 3)
>>> torch.zeros_like(input)
tensor([[ 0.,  0.,  0.],
        [ 0.,  0.,  0.]])
```

`torch.`**`ones`**(**sizes*, *out=None*, *dtype=None*, *layout=torch.strided*, *device=None*, *requires_grad=False*) → Tensor

Returns a tensor filled with the scalar value *1*, with the shape defined by the variable argument `sizes`.

Parameters

- **sizes** (*int...*) – a sequence of integers defining the shape of the output tensor. Can be a variable number of arguments or a collection like a list or tuple.
- **out** (Tensor, *optional*) – the output tensor
- **dtype** (*torch.dtype*, optional) – the desired data type of returned tensor. Default: if None, uses a global default (see *torch.set_default_tensor_type()*).
- **layout** (*torch.layout*, optional) – the desired layout of returned Tensor. Default: torch.strided.
- **device** (*torch.device*, optional) – the desired device of returned tensor. Default: if None, uses the current device for the default tensor type (see *torch.set_default_tensor_type()*). device will be the CPU for CPU tensor types and the current CUDA device for CUDA tensor types.
- **requires_grad** (*bool, optional*) – If autograd should record operations on the returned tensor. Default: False.

Example:

```
>>> torch.ones(2, 3)
tensor([[ 1.,  1.,  1.],
        [ 1.,  1.,  1.]])

>>> torch.ones(5)
tensor([ 1.,  1.,  1.,  1.,  1.])
```

torch.**ones_like**(*input*, *dtype=None*, *layout=None*, *device=None*, *requires_grad=False*) → Tensor

Returns a tensor filled with the scalar value *1*, with the same size as input. torch.ones_like(input) is equivalent to torch.ones(input.size(), dtype=input.dtype, layout=input.layout, device=input.device).

Warning: As of 0.4, this function does not support an out keyword. As an alternative, the old torch.ones_like(input, out=output) is equivalent to torch.ones(input.size(), out=output).

Parameters

- **input** (Tensor) – the size of input will determine size of the output tensor
- **dtype** (*torch.dtype*, optional) – the desired data type of returned Tensor. Default: if None, defaults to the dtype of input.
- **layout** (*torch.layout*, optional) – the desired layout of returned tensor. Default: if None, defaults to the layout of input.
- **device** (*torch.device*, optional) – the desired device of returned tensor. Default: if None, defaults to the device of input.
- **requires_grad** (*bool, optional*) – If autograd should record operations on the returned tensor. Default: False.

Example:

```
>>> input = torch.empty(2, 3)
>>> torch.ones_like(input)
tensor([[ 1.,  1.,  1.],
        [ 1.,  1.,  1.]])
```

torch.**arange**(*start=0*, *end*, *step=1*, *out=None*, *dtype=None*, *layout=torch.strided*, *device=None*, *requires_grad=False*) → Tensor

Returns a 1-D tensor of size $\left\lfloor \frac{\text{end} - \text{start}}{\text{step}} \right\rfloor$ with values from the interval `[start, end)` taken with common difference `step` beginning from *start*.

Note that non-integer `step` is subject to floating point rounding errors when comparing against `end`; to avoid inconsistency, we advise adding a small epsilon to `end` in such cases.

$$\text{out}_{i+1} = \text{out}_i + \text{step}$$

Parameters

- **start** (*Number*) – the starting value for the set of points. Default: `0`.
- **end** (*Number*) – the ending value for the set of points
- **step** (*Number*) – the gap between each pair of adjacent points. Default: `1`.
- **out** (*Tensor, optional*) – the output tensor
- **dtype** (*torch.dtype*, optional) – the desired data type of returned tensor. Default: if `None`, uses a global default (see *torch.set_default_tensor_type()*). If *dtype* is not given, infer the data type from the other input arguments. If any of *start*, *end*, or *stop* are floating-point, the *dtype* is inferred to be the default dtype, see *get_default_dtype()*. Otherwise, the *dtype* is inferred to be *torch.int64*.
- **layout** (*torch.layout*, optional) – the desired layout of returned Tensor. Default: `torch.strided`.
- **device** (*torch.device*, optional) – the desired device of returned tensor. Default: if `None`, uses the current device for the default tensor type (see *torch.set_default_tensor_type()*). `device` will be the CPU for CPU tensor types and the current CUDA device for CUDA tensor types.
- **requires_grad** (*bool, optional*) – If autograd should record operations on the returned tensor. Default: `False`.

Example:

```
>>> torch.arange(5)
tensor([ 0,  1,  2,  3,  4])
>>> torch.arange(1, 4)
tensor([ 1,  2,  3])
>>> torch.arange(1, 2.5, 0.5)
tensor([ 1.0000,  1.5000,  2.0000])
```

torch.**range**(*start=0*, *end*, *step=1*, *out=None*, *dtype=None*, *layout=torch.strided*, *device=None*, *requires_grad=False*) → Tensor

Returns a 1-D tensor of size $\left\lfloor \frac{\text{end} - \text{start}}{\text{step}} \right\rfloor + 1$ with values from `start` to `end` with step `step`. Step is the gap between two values in the tensor.

$$\text{out}_{i+1} = \text{out}_i + \text{step}.$$

Warning: This function is deprecated in favor of *torch.arange()*.

Parameters

- **start** (*float*) – the starting value for the set of points. Default: `0`.

- **end** (*float*) – the ending value for the set of points
- **step** (*float*) – the gap between each pair of adjacent points. Default: `1`.
- **out** (*Tensor, optional*) – the output tensor
- **dtype** (*torch.dtype*, optional) – the desired data type of returned tensor. Default: if `None`, uses a global default (see *torch.set_default_tensor_type()*). If *dtype* is not given, infer the data type from the other input arguments. If any of *start*, *end*, or *stop* are floating-point, the *dtype* is inferred to be the default dtype, see *get_default_dtype()*. Otherwise, the *dtype* is inferred to be *torch.int64*.
- **layout** (*torch.layout*, optional) – the desired layout of returned Tensor. Default: `torch.strided`.
- **device** (*torch.device*, optional) – the desired device of returned tensor. Default: if `None`, uses the current device for the default tensor type (see *torch.set_default_tensor_type()*). `device` will be the CPU for CPU tensor types and the current CUDA device for CUDA tensor types.
- **requires_grad** (*bool, optional*) – If autograd should record operations on the returned tensor. Default: `False`.

Example:

```
>>> torch.range(1, 4)
tensor([ 1.,  2.,  3.,  4.])
>>> torch.range(1, 4, 0.5)
tensor([ 1.0000,  1.5000,  2.0000,  2.5000,  3.0000,  3.5000,  4.0000])
```

torch.**linspace**(*start*, *end*, *steps=100*, *out=None*, *dtype=None*, *layout=torch.strided*, *device=None*, *requires_grad=False*) → Tensor

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

The output tensor is 1-D of size `steps`.

Parameters

- **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`. Default: `100`.
- **out** (*Tensor, optional*) – the output tensor
- **dtype** (*torch.dtype*, optional) – the desired data type of returned tensor. Default: if `None`, uses a global default (see *torch.set_default_tensor_type()*).
- **layout** (*torch.layout*, optional) – the desired layout of returned Tensor. Default: `torch.strided`.
- **device** (*torch.device*, optional) – the desired device of returned tensor. Default: if `None`, uses the current device for the default tensor type (see *torch.set_default_tensor_type()*). `device` will be the CPU for CPU tensor types and the current CUDA device for CUDA tensor types.
- **requires_grad** (*bool, optional*) – If autograd should record operations on the returned tensor. Default: `False`.

Example:

```
>>> torch.linspace(3, 10, steps=5)
tensor([  3.0000,   4.7500,   6.5000,   8.2500,  10.0000])
>>> torch.linspace(-10, 10, steps=5)
tensor([-10.,  -5.,   0.,   5.,  10.])
>>> torch.linspace(start=-10, end=10, steps=5)
tensor([-10.,  -5.,   0.,   5.,  10.])
>>> torch.linspace(start=-10, end=10, steps=1)
tensor([-10.])
```

torch.**logspace**(*start*, *end*, *steps=100*, *out=None*, *dtype=None*, *layout=torch.strided*, *device=None*, *requires_grad=False*) → Tensor

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

The output tensor is 1-D of size `steps`.

Parameters

- **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`. Default: `100`.
- **out** (Tensor, *optional*) – the output tensor
- **dtype** (*torch.dtype*, optional) – the desired data type of returned tensor. Default: if `None`, uses a global default (see *torch.set_default_tensor_type()*).
- **layout** (*torch.layout*, optional) – the desired layout of returned Tensor. Default: `torch.strided`.
- **device** (*torch.device*, optional) – the desired device of returned tensor. Default: if `None`, uses the current device for the default tensor type (see *torch.set_default_tensor_type()*). `device` will be the CPU for CPU tensor types and the current CUDA device for CUDA tensor types.
- **requires_grad** (*bool*, *optional*) – If autograd should record operations on the returned tensor. Default: `False`.

Example:

```
>>> torch.logspace(start=-10, end=10, steps=5)
tensor([ 1.0000e-10,  1.0000e-05,  1.0000e+00,  1.0000e+05,  1.0000e+10])
>>> torch.logspace(start=0.1, end=1.0, steps=5)
tensor([  1.2589,   2.1135,   3.5481,   5.9566,  10.0000])
>>> torch.logspace(start=0.1, end=1.0, steps=1)
tensor([1.2589])
```

torch.**eye**(*n*, *m=None*, *out=None*, *dtype=None*, *layout=torch.strided*, *device=None*, *requires_grad=False*) → Tensor

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

Parameters

- **n** (*int*) – the number of rows
- **m** (*int*, *optional*) – the number of columns with default being `n`
- **out** (Tensor, *optional*) – the output tensor
- **dtype** (*torch.dtype*, optional) – the desired data type of returned tensor. Default: if `None`, uses a global default (see *torch.set_default_tensor_type()*).

- **layout** (*torch.layout*, optional) – the desired layout of returned Tensor. Default: `torch.strided`.
- **device** (*torch.device*, optional) – the desired device of returned tensor. Default: if `None`, uses the current device for the default tensor type (see *torch.set_default_tensor_type()*). `device` will be the CPU for CPU tensor types and the current CUDA device for CUDA tensor types.
- **requires_grad** (*bool, optional*) – If autograd should record operations on the returned tensor. Default: `False`.

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

Return type *Tensor*

Example:

```
>>> torch.eye(3)
tensor([[ 1.,  0.,  0.],
        [ 0.,  1.,  0.],
        [ 0.,  0.,  1.]])
```

`torch.`**`empty`**(**sizes*, *out=None*, *dtype=None*, *layout=torch.strided*, *device=None*, *requires_grad=False*, *pin_memory=False*) → Tensor

Returns a tensor filled with uninitialized data. The shape of the tensor is defined by the variable argument `sizes`.

Parameters

- **sizes** (*int...*) – a sequence of integers defining the shape of the output tensor. Can be a variable number of arguments or a collection like a list or tuple.
- **out** (*Tensor, optional*) – the output tensor
- **dtype** (*torch.dtype*, optional) – the desired data type of returned tensor. Default: if `None`, uses a global default (see *torch.set_default_tensor_type()*).
- **layout** (*torch.layout*, optional) – the desired layout of returned Tensor. Default: `torch.strided`.
- **device** (*torch.device*, optional) – the desired device of returned tensor. Default: if `None`, uses the current device for the default tensor type (see *torch.set_default_tensor_type()*). `device` will be the CPU for CPU tensor types and the current CUDA device for CUDA tensor types.
- **requires_grad** (*bool, optional*) – If autograd should record operations on the returned tensor. Default: `False`.
- **pin_memory** (*bool, optional*) – If set, returned tensor would be allocated in the pinned memory. Works only for CPU tensors. Default: `False`.

Example:

```
>>> torch.empty(2, 3)
tensor(1.00000e-08 *
       [[ 6.3984,  0.0000,  0.0000],
        [ 0.0000,  0.0000,  0.0000]])
```

`torch.`**`empty_like`**(*input*, *dtype=None*, *layout=None*, *device=None*, *requires_grad=False*) → Tensor

Returns an uninitialized tensor with the same size as `input`. `torch.empty_like(input)` is equivalent to `torch.empty(input.size(), dtype=input.dtype, layout=input.layout, device=input.device)`.

Parameters

- **input** (Tensor) – the size of input will determine size of the output tensor
- **dtype** (*torch.dtype*, optional) – the desired data type of returned Tensor. Default: if None, defaults to the dtype of input.
- **layout** (*torch.layout*, optional) – the desired layout of returned tensor. Default: if None, defaults to the layout of input.
- **device** (*torch.device*, optional) – the desired device of returned tensor. Default: if None, defaults to the device of input.
- **requires_grad** (*bool, optional*) – If autograd should record operations on the returned tensor. Default: False.

Example:

```
>>> torch.empty((2,3), dtype=torch.int64)
tensor([[ 9.4064e+13,  2.8000e+01,  9.3493e+13],
        [ 7.5751e+18,  7.1428e+18,  7.5955e+18]])
```

torch.**empty_strided**()

torch.**full**(*size*, *fill_value*, *out=None*, *dtype=None*, *layout=torch.strided*, *device=None*, *requires_grad=False*) → Tensor

Returns a tensor of size size filled with fill_value.

Parameters

- **size** (*int...*) – a list, tuple, or torch.Size of integers defining the shape of the output tensor.
- **fill_value** – the number to fill the output tensor with.
- **out** (Tensor, *optional*) – the output tensor
- **dtype** (*torch.dtype*, optional) – the desired data type of returned tensor. Default: if None, uses a global default (see *torch.set_default_tensor_type()*).
- **layout** (*torch.layout*, optional) – the desired layout of returned Tensor. Default: torch.strided.
- **device** (*torch.device*, optional) – the desired device of returned tensor. Default: if None, uses the current device for the default tensor type (see *torch.set_default_tensor_type()*). device will be the CPU for CPU tensor types and the current CUDA device for CUDA tensor types.
- **requires_grad** (*bool, optional*) – If autograd should record operations on the returned tensor. Default: False.

Example:

```
>>> torch.full((2, 3), 3.141592)
tensor([[ 3.1416,  3.1416,  3.1416],
        [ 3.1416,  3.1416,  3.1416]])
```

torch.**full_like**(*input*, *fill_value*, *out=None*, *dtype=None*, *layout=torch.strided*, *device=None*, *requires_grad=False*) → Tensor

Returns a tensor with the same size as input filled with fill_value. torch.full_like(input, fill_value) is equivalent to torch.full(input.size(), fill_value, dtype=input.dtype, layout=input.layout, device=input.device).

Parameters

- **input** (Tensor) – the size of `input` will determine size of the output tensor
- **fill_value** – the number to fill the output tensor with.
- **dtype** (*torch.dtype*, optional) – the desired data type of returned Tensor. Default: if `None`, defaults to the dtype of `input`.
- **layout** (*torch.layout*, optional) – the desired layout of returned tensor. Default: if `None`, defaults to the layout of `input`.
- **device** (*torch.device*, optional) – the desired device of returned tensor. Default: if `None`, defaults to the device of `input`.
- **requires_grad** (*bool, optional*) – If autograd should record operations on the returned tensor. Default: `False`.

15.1.2 Indexing, Slicing, Joining, Mutating Ops

`torch.`**`cat`**(*tensors*, *dim=0*, *out=None*) → Tensor

Concatenates the given sequence of `seq` tensors in the given dimension. All tensors must either have the same shape (except in the concatenating dimension) or be empty.

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

torch.cat() can be best understood via examples.

Parameters

- **tensors** (*sequence of Tensors*) – any python sequence of tensors of the same type. Non-empty tensors provided must have the same shape, except in the cat dimension.
- **dim** (*int, optional*) – the dimension over which the tensors are concatenated
- **out** (Tensor, *optional*) – the output tensor

Example:

```
>>> x = torch.randn(2, 3)
>>> x
tensor([[ 0.6580, -1.0969, -0.4614],
        [-0.1034, -0.5790,  0.1497]])
>>> torch.cat((x, x, x), 0)
tensor([[ 0.6580, -1.0969, -0.4614],
        [-0.1034, -0.5790,  0.1497],
        [ 0.6580, -1.0969, -0.4614],
        [-0.1034, -0.5790,  0.1497],
        [ 0.6580, -1.0969, -0.4614],
        [-0.1034, -0.5790,  0.1497]])
>>> torch.cat((x, x, x), 1)
tensor([[ 0.6580, -1.0969, -0.4614,  0.6580, -1.0969, -0.4614,  0.6580,
         -1.0969, -0.4614],
        [-0.1034, -0.5790,  0.1497, -0.1034, -0.5790,  0.1497, -0.1034,
         -0.5790,  0.1497]])
```

`torch.`**`chunk`**(*tensor*, *chunks*, *dim=0*) → List of Tensors

Splits a tensor into a specific number of chunks.

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

Parameters

- **tensor** (Tensor) – the 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*, *sparse_grad=False*) → Tensor

Gathers values along an axis specified by *dim*.

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

```
out[i][j][k] = input[index[i][j][k]][j][k]  # if dim == 0
out[i][j][k] = input[i][index[i][j][k]][k]  # if dim == 1
out[i][j][k] = input[i][j][index[i][j][k]]  # if dim == 2
```

If `input` is an n-dimensional tensor with size $(x_0, x_1..., x_{i-1}, x_i, x_{i+1}, ..., x_{n-1})$ and `dim = i`, then `index` must be an n-dimensional tensor with size $(x_0, x_1, ..., x_{i-1}, y, x_{i+1}, ..., x_{n-1})$ where $y \geq 1$ and `out` will have the same size as `index`.

Parameters

- **input** (*Tensor*) – the source tensor
- **dim** (*int*) – the axis along which to index
- **index** (*LongTensor*) – the indices of elements to gather
- **out** (*Tensor, optional*) – the destination tensor
- **sparse_grad** (*bool,optional*) – If `True`, gradient w.r.t. `input` will be a sparse tensor.

Example:

```
>>> t = torch.tensor([[1,2],[3,4]])
>>> torch.gather(t, 1, torch.tensor([[0,0],[1,0]]))
tensor([[ 1,  1],
        [ 4,  3]])
```

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 (`input`). The `dim`th dimension has the same size as the length of `index`; other dimensions have the same size as in the original tensor.

Note: The returned tensor does **not** use the same storage as the original tensor. If `out` has a different shape than expected, we silently change it to the correct shape, reallocating the underlying storage if necessary.

Parameters

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

Example:

```
>>> x = torch.randn(3, 4)
>>> x
tensor([[ 0.1427,  0.0231, -0.5414, -1.0009],
        [-0.4664,  0.2647, -0.1228, -1.1068],
        [-1.1734, -0.6571,  0.7230, -0.6004]])
>>> indices = torch.tensor([0, 2])
>>> torch.index_select(x, 0, indices)
tensor([[ 0.1427,  0.0231, -0.5414, -1.0009],
        [-1.1734, -0.6571,  0.7230, -0.6004]])
>>> torch.index_select(x, 1, indices)
tensor([[ 0.1427, -0.5414],
        [-0.4664, -0.1228],
        [-1.1734,  0.7230]])
```

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

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

The shapes of the `mask` tensor and the `input` tensor dont need to match, but they must be *broadcastable*.

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

---

Parameters

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

Example:

```
>>> x = torch.randn(3, 4)
>>> x
tensor([[ 0.3552, -2.3825, -0.8297,  0.3477],
        [-1.2035,  1.2252,  0.5002,  0.6248],
        [ 0.1307, -2.0608,  0.1244,  2.0139]])
>>> mask = x.ge(0.5)
>>> mask
tensor([[ 0,  0,  0,  0],
        [ 0,  1,  1,  1],
        [ 0,  0,  0,  1]], dtype=torch.uint8)
>>> torch.masked_select(x, mask)
tensor([ 1.2252,  0.5002,  0.6248,  2.0139])
```

torch.**narrow**(*input*, *dimension*, *start*, *length*) → Tensor

Returns a new tensor that is a narrowed version of `input` tensor. The dimension `dim` is input from `start` to `start + length`. The returned tensor and `input` tensor share the same underlying storage.

Parameters

- **input** (Tensor) – the tensor to narrow
- **dimension** (*int*) – the dimension along which to narrow
- **start** (*int*) – the starting dimension
- **length** (*int*) – the distance to the ending dimension

Example:

```
>>> x = torch.tensor([[1, 2, 3], [4, 5, 6], [7, 8, 9]])
>>> torch.narrow(x, 0, 0, 2)
tensor([[ 1,  2,  3],
        [ 4,  5,  6]])
>>> torch.narrow(x, 1, 1, 2)
tensor([[ 2,  3],
        [ 5,  6],
        [ 8,  9]])
```

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 $(z \times n)$, where z is the total number of non-zero elements in the `input` tensor.

Parameters

- **input** (Tensor) – the input tensor
- **out** (*LongTensor, optional*) – the output tensor containing indices

Example:

```
>>> torch.nonzero(torch.tensor([1, 1, 1, 0, 1]))
tensor([[ 0],
        [ 1],
        [ 2],
        [ 4]])
>>> 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]]))
tensor([[ 0,  0],
        [ 1,  1],
        [ 2,  2],
        [ 3,  3]])
```

torch.**reshape**(*input*, *shape*) → Tensor

Returns a tensor with the same data and number of elements as `input`, but with the specified shape. When possible, the returned tensor will be a view of `input`. Otherwise, it will be a copy. Contiguous inputs and inputs with compatible strides can be reshaped without copying, but you should not depend on the copying vs. viewing behavior.

See *torch.Tensor.view()* on when it is possible to return a view.

A single dimension may be -1, in which case its inferred from the remaining dimensions and the number of elements in `input`.

Parameters

- **input** (Tensor) – the tensor to be reshaped
- **shape** (*tuple of python:ints*) – the new shape

Example:

```
>>> a = torch.arange(4.)
>>> torch.reshape(a, (2, 2))
```

(continues on next page)

(continued from previous page)

```
tensor([[ 0.,  1.],
        [ 2.,  3.]])
>>> b = torch.tensor([[0, 1], [2, 3]])
>>> torch.reshape(b, (-1,))
tensor([ 0,  1,  2,  3])
```

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

Splits the tensor into chunks.

If `split_size_or_sections` is an integer type, then *tensor* will be split into equally sized chunks (if possible). Last chunk will be smaller if the tensor size along the given dimension `dim` is not divisible by `split_size`.

If `split_size_or_sections` is a list, then *tensor* will be split into `len(split_size_or_sections)` chunks with sizes in `dim` according to `split_size_or_sections`.

Parameters

- **tensor** (Tensor) – tensor to split.
- **split_size_or_sections** (*int) or (list(int)*) – size of a single chunk or list of sizes for each chunk
- **dim** (*int*) – dimension along which to split the tensor.

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

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

For example, if *input* is of shape: $(A \times 1 \times B \times C \times 1 \times D)$ then the *out* tensor will be of shape: $(A \times B \times C \times D)$.

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

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

Parameters

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

Example:

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

(continues on next page)

(continued from previous page)

```
>>> y.size()
torch.Size([2, 2, 1, 2])
```

torch.**stack**(*seq*, *dim=0*, *out=None*) → Tensor

Concatenates sequence of tensors along a new dimension.

All tensors need to be of the same size.

Parameters

- **seq** (*sequence of Tensors*) – sequence of tensors to concatenate
- **dim** (*int*) – dimension to insert. Has to be between 0 and the number of dimensions of concatenated tensors (inclusive)
- **out** (Tensor, *optional*) – the output tensor

torch.**t**(*input*) → Tensor

Expects `input` to be <= 2-D tensor and transposes dimensions 0 and 1.

0-D and 1-D tensors are returned as it is and 2-D tensor can be seen as a short-hand function for `transpose(input, 0, 1)`.

Parameters **input** (Tensor) – the input tensor

Example:

```
>>> x = torch.randn(())
>>> x
tensor(0.1995)
>>> torch.t(x)
tensor(0.1995)
>>> x = torch.randn(3)
>>> x
tensor([ 2.4320, -0.4608,  0.7702])
>>> torch.t(x)
tensor([.2.4320,.-0.4608,..0.7702])
>>> x = torch.randn(2, 3)
>>> x
tensor([[ 0.4875,  0.9158, -0.5872],
        [ 0.3938, -0.6929,  0.6932]])
>>> torch.t(x)
tensor([[ 0.4875,  0.3938],
        [ 0.9158, -0.6929],
        [-0.5872,  0.6932]])
```

torch.**take**(*input*, *indices*) → Tensor

Returns a new tensor with the elements of `input` at the given indices. The input tensor is treated as if it were viewed as a 1-D tensor. The result takes the same shape as the indices.

Parameters

- **input** (Tensor) – the input tensor
- **indices** (*LongTensor*) – the indices into tensor

Example:

```
>>> src = torch.tensor([[4, 3, 5],
                        [6, 7, 8]])
```

(continues on next page)

(continued from previous page)

```
>>> torch.take(src, torch.tensor([0, 2, 5]))
tensor([ 4,  5,  8])
```

torch.**transpose**(*input*, *dim0*, *dim1*) → Tensor

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

The resulting `out` tensor shares its underlying storage with the `input` tensor, so changing the content of one would change the content of the other.

Parameters

- **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
tensor([[ 1.0028, -0.9893,  0.5809],
        [-0.1669,  0.7299,  0.4942]])
>>> torch.transpose(x, 0, 1)
tensor([[ 1.0028, -0.1669],
        [-0.9893,  0.7299],
        [ 0.5809,  0.4942]])
```

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

Removes a tensor dimension.

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

Parameters

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

Example:

```
>>> torch.unbind(torch.tensor([[1, 2, 3],
>>>                            [4, 5, 6],
>>>                            [7, 8, 9]]))
(tensor([1, 2, 3]), tensor([4, 5, 6]), tensor([7, 8, 9]))
```

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

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.

A `dim` value within the range `[-input.dim() - 1, input.dim() + 1)` can be used. Negative `dim` will correspond to *`unsqueeze()`* applied at `dim = dim + input.dim() + 1`.

Parameters

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

Example:

```
>>> x = torch.tensor([1, 2, 3, 4])
>>> torch.unsqueeze(x, 0)
tensor([[ 1,  2,  3,  4]])
>>> torch.unsqueeze(x, 1)
tensor([[ 1],
        [ 2],
        [ 3],
        [ 4]])
```

torch.**where**(*condition*, *x*, *y*) → Tensor

Return a tensor of elements selected from either `x` or `y`, depending on `condition`.

The operation is defined as:

$$out_i = \begin{cases} x_i & \text{if condition}_i \\ y_i & \text{otherwise} \end{cases}$$

Note: The tensors `condition`, `x`, `y` must be *broadcastable*.

Parameters

- **condition** (*ByteTensor*) – When True (nonzero), yield x, otherwise yield y
- **x** (Tensor) – values selected at indices where `condition` is `True`
- **y** (Tensor) – values selected at indices where `condition` is `False`

Returns A tensor of shape equal to the broadcasted shape of `condition`, `x`, `y`

Return type *Tensor*

Example:

```
>>> x = torch.randn(3, 2)
>>> y = torch.ones(3, 2)
>>> x
tensor([[-0.4620,  0.3139],
        [ 0.3898, -0.7197],
        [ 0.0478, -0.1657]])
>>> torch.where(x > 0, x, y)
tensor([[ 1.0000,  0.3139],
        [ 0.3898,  1.0000],
        [ 0.0478,  1.0000]])
```

15.2 Generators

class torch._C.**Generator**

15.3 Random sampling

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

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

Parameters **seed** (*int*) – 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 *torch.ByteTensor*.

torch.**set_rng_state**(*new_state*)
Sets the random number generator state.

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

torch.**default_generator Returns the default CPU torch.Generator**

torch.**bernoulli**(*input*, *, *generator=None*, *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: $0 \leq \text{input}_i \leq 1$.

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

$$\text{out}_i \sim \text{Bernoulli}(p = \text{input}_i)$$

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

`out` can have integral `dtype`, but `input` must have floating point `dtype`.

Parameters

- **input** (Tensor) – the input tensor of probability values for the Bernoulli distribution
- **out** (Tensor, *optional*) – the output tensor

Example:

```
>>> a = torch.empty(3, 3).uniform_(0, 1)  # generate a uniform random matrix with␣
↪range [0, 1]
>>> a
tensor([[ 0.1737,  0.0950,  0.3609],
        [ 0.7148,  0.0289,  0.2676],
        [ 0.9456,  0.8937,  0.7202]])
>>> torch.bernoulli(a)
tensor([[ 1.,  0.,  0.],
        [ 0.,  0.,  0.],
        [ 1.,  1.,  1.]])

>>> a = torch.ones(3, 3) # probability of drawing "1" is 1
>>> torch.bernoulli(a)
tensor([[ 1.,  1.,  1.],
        [ 1.,  1.,  1.],
        [ 1.,  1.,  1.]])
>>> a = torch.zeros(3, 3) # probability of drawing "1" is 0
>>> torch.bernoulli(a)
tensor([[ 0.,  0.,  0.],
        [ 0.,  0.,  0.],
        [ 0.,  0.,  0.]])
```

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`.

Note: 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, finite 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 $(m \times \text{num\_samples})$.

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.

Note: When drawn without replacement, `num_samples` must be lower than number of non-zero elements in `input` (or the min number of non-zero elements in each row of `input` if it is a matrix).

Parameters

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

Example:

```
>>> weights = torch.tensor([0, 10, 3, 0], dtype=torch.float) # create a tensor of␣
↪weights
>>> torch.multinomial(weights, 2)
tensor([1, 2])
>>> torch.multinomial(weights, 4) # ERROR!
RuntimeError: invalid argument 2: invalid multinomial distribution (with␣
↪replacement=False,
not enough non-negative category to sample) at ../aten/src/TH/generic/
↪THTensorRandom.cpp:320
>>> torch.multinomial(weights, 4, replacement=True)
tensor([ 2,  1,  1,  1])
```

`torch.`**`normal`**`()`

`torch.`**`normal`**(*mean*, *std*, *out=None*) → Tensor

Returns a tensor of random numbers drawn from separate normal distributions whose mean and standard deviation are given.

The *mean* is a tensor with the mean of each output elements normal distribution

The *std* is a tensor with the standard deviation of each output elements normal distribution

The shapes of *mean* and *std* dont need to match, but the total number of elements in each tensor need to be the same.

Note: When the shapes do not match, the shape of *mean* is used as the shape for the returned output tensor

Parameters

- **mean** (Tensor) – the tensor of per-element means
- **std** (Tensor) – the tensor of per-element standard deviations
- **out** (Tensor, *optional*) – the output tensor

Example:

```
>>> torch.normal(mean=torch.arange(1., 11.), std=torch.arange(1, 0, -0.1))
tensor([  1.0425,   3.5672,   2.7969,   4.2925,   4.7229,   6.2134,
          8.0505,   8.1408,   9.0563,  10.0566])
```

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

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

Parameters

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

Example:

```
>>> torch.normal(mean=0.5, std=torch.arange(1., 6.))
tensor([-1.2793, -1.0732, -2.0687,  5.1177, -1.2303])
```

torch.**normal**(*mean*, *std=1.0*, *out=None*) → Tensor

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

Parameters

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

Example:

```
>>> torch.normal(mean=torch.arange(1., 6.))
tensor([ 1.1552,  2.6148,  2.6535,  5.8318,  4.2361])
```

torch.**rand**(**sizes*, *out=None*, *dtype=None*, *layout=torch.strided*, *device=None*, *requires_grad=False*) → 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 variable argument `sizes`.

Parameters

- **sizes** (*int...*) – a sequence of integers defining the shape of the output tensor. Can be a variable number of arguments or a collection like a list or tuple.
- **out** (Tensor, *optional*) – the output tensor

- **dtype** (*torch.dtype*, optional) – the desired data type of returned tensor. Default: if None, uses a global default (see *torch.set_default_tensor_type()*).
- **layout** (*torch.layout*, optional) – the desired layout of returned Tensor. Default: torch.strided.
- **device** (*torch.device*, optional) – the desired device of returned tensor. Default: if None, uses the current device for the default tensor type (see *torch.set_default_tensor_type()*). device will be the CPU for CPU tensor types and the current CUDA device for CUDA tensor types.
- **requires_grad** (*bool, optional*) – If autograd should record operations on the returned tensor. Default: False.

Example:

```
>>> torch.rand(4)
tensor([ 0.5204,  0.2503,  0.3525,  0.5673])
>>> torch.rand(2, 3)
tensor([[ 0.8237,  0.5781,  0.6879],
        [ 0.3816,  0.7249,  0.0998]])
```

torch.**rand_like**(*input*, *dtype=None*, *layout=None*, *device=None*, *requires_grad=False*) → Tensor

Returns a tensor with the same size as input that is filled with random numbers from a uniform distribution on the interval $[0, 1)$. torch.rand_like(input) is equivalent to torch.rand(input.size(), dtype=input.dtype, layout=input.layout, device=input.device).

Parameters

- **input** (Tensor) – the size of input will determine size of the output tensor
- **dtype** (*torch.dtype*, optional) – the desired data type of returned Tensor. Default: if None, defaults to the dtype of input.
- **layout** (*torch.layout*, optional) – the desired layout of returned tensor. Default: if None, defaults to the layout of input.
- **device** (*torch.device*, optional) – the desired device of returned tensor. Default: if None, defaults to the device of input.
- **requires_grad** (*bool, optional*) – If autograd should record operations on the returned tensor. Default: False.

torch.**randint**(*low=0*, *high*, *size*, *out=None*, *dtype=None*, *layout=torch.strided*, *device=None*, *requires_grad=False*) → Tensor

Returns a tensor filled with random integers generated uniformly between low (inclusive) and high (exclusive).

The shape of the tensor is defined by the variable argument size.

Parameters

- **low** (*int, optional*) – Lowest integer to be drawn from the distribution. Default: 0.
- **high** (*int*) – One above the highest integer to be drawn from the distribution.
- **size** (*tuple*) – a tuple defining the shape of the output tensor.
- **out** (Tensor, *optional*) – the output tensor
- **dtype** (*torch.dtype*, optional) – the desired data type of returned tensor. Default: if None, uses a global default (see *torch.set_default_tensor_type()*).
- **layout** (*torch.layout*, optional) – the desired layout of returned Tensor. Default: torch.strided.

- **device** (*torch.device*, optional) – the desired device of returned tensor. Default: if `None`, uses the current device for the default tensor type (see *torch.set_default_tensor_type()*). `device` will be the CPU for CPU tensor types and the current CUDA device for CUDA tensor types.
- **requires_grad** (*bool, optional*) – If autograd should record operations on the returned tensor. Default: `False`.

Example:

```
>>> torch.randint(3, 5, (3,))
tensor([4, 3, 4])


>>> torch.randint(10, (2, 2))
tensor([[0, 2],
        [5, 5]])


>>> torch.randint(3, 10, (2, 2))
tensor([[4, 5],
        [6, 7]])
```

`torch.`**`randint_like`**(*input*, *low=0*, *high*, *dtype=None*, *layout=torch.strided*, *device=None*, *requires_grad=False*) → Tensor

Returns a tensor with the same shape as Tensor `input` filled with random integers generated uniformly between `low` (inclusive) and `high` (exclusive).

Parameters

- **input** (`Tensor`) – the size of `input` will determine size of the output tensor
- **low** (*int, optional*) – Lowest integer to be drawn from the distribution. Default: 0.
- **high** (*int*) – One above the highest integer to be drawn from the distribution.
- **dtype** (*torch.dtype*, optional) – the desired data type of returned Tensor. Default: if `None`, defaults to the dtype of `input`.
- **layout** (*torch.layout*, optional) – the desired layout of returned tensor. Default: if `None`, defaults to the layout of `input`.
- **device** (*torch.device*, optional) – the desired device of returned tensor. Default: if `None`, defaults to the device of `input`.
- **requires_grad** (*bool, optional*) – If autograd should record operations on the returned tensor. Default: `False`.

`torch.`**`randn`**(**sizes*, *out=None*, *dtype=None*, *layout=torch.strided*, *device=None*, *requires_grad=False*) → Tensor

Returns a tensor filled with random numbers from a normal distribution with mean *0* and variance *1* (also called the standard normal distribution).

$$\text{out}_i \sim \mathcal{N}(0, 1)$$

The shape of the tensor is defined by the variable argument `sizes`.

Parameters

- **sizes** (*int...*) – a sequence of integers defining the shape of the output tensor. Can be a variable number of arguments or a collection like a list or tuple.
- **out** (`Tensor, optional`) – the output tensor

- **dtype** (*torch.dtype*, optional) – the desired data type of returned tensor. Default: if None, uses a global default (see *torch.set_default_tensor_type()*).
- **layout** (*torch.layout*, optional) – the desired layout of returned Tensor. Default: torch.strided.
- **device** (*torch.device*, optional) – the desired device of returned tensor. Default: if None, uses the current device for the default tensor type (see *torch.set_default_tensor_type()*). device will be the CPU for CPU tensor types and the current CUDA device for CUDA tensor types.
- **requires_grad** (*bool, optional*) – If autograd should record operations on the returned tensor. Default: False.

Example:

```
>>> torch.randn(4)
tensor([-2.1436,  0.9966,  2.3426, -0.6366])
>>> torch.randn(2, 3)
tensor([[ 1.5954,  2.8929, -1.0923],
        [ 1.1719, -0.4709, -0.1996]])
```

torch.**randn_like**(*input*, *dtype=None*, *layout=None*, *device=None*, *requires_grad=False*) → Tensor

Returns a tensor with the same size as input that is filled with random numbers from a normal distribution with mean 0 and variance 1. torch.randn_like(input) is equivalent to torch.randn(input.size(), dtype=input.dtype, layout=input.layout, device=input.device).

Parameters

- **input** (Tensor) – the size of input will determine size of the output tensor
- **dtype** (*torch.dtype*, optional) – the desired data type of returned Tensor. Default: if None, defaults to the dtype of input.
- **layout** (*torch.layout*, optional) – the desired layout of returned tensor. Default: if None, defaults to the layout of input.
- **device** (*torch.device*, optional) – the desired device of returned tensor. Default: if None, defaults to the device of input.
- **requires_grad** (*bool, optional*) – If autograd should record operations on the returned tensor. Default: False.

torch.**randperm**(*n*, *out=None*, *dtype=torch.int64*, *layout=torch.strided*, *device=None*, *requires_grad=False*) → LongTensor

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

Parameters

- **n** (*int*) – the upper bound (exclusive)
- **out** (Tensor, *optional*) – the output tensor
- **dtype** (*torch.dtype*, optional) – the desired data type of returned tensor. Default: torch.int64.
- **layout** (*torch.layout*, optional) – the desired layout of returned Tensor. Default: torch.strided.
- **device** (*torch.device*, optional) – the desired device of returned tensor. Default: if None, uses the current device for the default tensor type (see *torch.set_default_tensor_type()*). device will be the CPU for CPU tensor types and the current CUDA device for CUDA tensor types.

- **requires_grad** (*bool, optional*) – If autograd should record operations on the returned tensor. Default: `False`.

Example:

```
>>> torch.randperm(4)
tensor([2, 1, 0, 3])
```

15.3.1 In-place random sampling

There are a few more in-place random sampling functions defined on Tensors as well. Click through to refer to their documentation:

- *torch.Tensor.bernoulli_()* - in-place version of *torch.bernoulli()*
- *torch.Tensor.cauchy_()* - numbers drawn from the Cauchy distribution
- *torch.Tensor.exponential_()* - numbers drawn from the exponential distribution
- *torch.Tensor.geometric_()* - elements drawn from the geometric distribution
- *torch.Tensor.log_normal_()* - samples from the log-normal distribution
- *torch.Tensor.normal_()* - in-place version of *torch.normal()*
- *torch.Tensor.random_()* - numbers sampled from the discrete uniform distribution
- *torch.Tensor.uniform_()* - numbers sampled from the continuous uniform distribution

15.3.2 Quasi-random sampling

class torch.quasirandom.**SobolEngine**(*dimension*, *scramble=False*, *seed=None*)

The *torch.quasirandom.SobolEngine* is an engine for generating (scrambled) Sobol sequences. Sobol sequences are an example of low discrepancy quasi-random sequences.

This implementation of an engine for Sobol sequences is capable of sampling sequences up to a maximum dimension of 1111. It uses direction numbers to generate these sequences, and these numbers have been adapted from here.

References

- Art B. Owen. Scrambling Sobol and Niederreiter-Xing points. Journal of Complexity, 14(4):466-489, December 1998.
- I. M. Sobol. The distribution of points in a cube and the accurate evaluation of integrals. Zh. Vychisl. Mat. i Mat. Phys., 7:784-802, 1967.

Parameters

- **dimension** (*Int*) – The dimensionality of the sequence to be drawn
- **scramble** (*bool, optional*) – Setting this to `True` will produce scrambled Sobol sequences. Scrambling is capable of producing better Sobol sequences. Default: `False`.
- **seed** (*Int, optional*) – This is the seed for the scrambling. The seed of the random number generator is set to this, if specified. Default: `None`

Examples:

```
>>> soboleng = torch.quasirandom.SobolEngine(dimension=5)
>>> soboleng.draw(3)
tensor([[0.5000, 0.5000, 0.5000, 0.5000, 0.5000],
        [0.7500, 0.2500, 0.7500, 0.2500, 0.7500],
        [0.2500, 0.7500, 0.2500, 0.7500, 0.2500]])
```

draw(*n=1*, *out=None*, *dtype=torch.float32*)

Function to draw a sequence of `n` points from a Sobol sequence. Note that the samples are dependent on the previous samples. The size of the result is $(n, dimension)$.

Parameters

- **n** (*Int, optional*) – The length of sequence of points to draw. Default: 1
- **out** (Tensor, *optional*) – The output tensor
- **dtype** (`torch.dtype`, optional) – the desired data type of the returned tensor. Default: `torch.float32`

fast_forward(*n*)

Function to fast-forward the state of the `SobolEngine` by `n` steps. This is equivalent to drawing `n` samples without using the samples.

Parameters **n** (*Int*) – The number of steps to fast-forward by.

reset()

Function to reset the `SobolEngine` to base state.

15.4 Serialization

torch.**save**(*obj*, *f*, *pickle_module=<module 'pickle' from '/home/yang/Essential/anaconda3/lib/python3.6/pickle.py'>*, *pickle_protocol=2*)

Saves an object to a disk file.

See also: *Recommended approach for saving a model*

Parameters

- **obj** – saved object
- **f** – a file-like object (has to implement write and flush) 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

> **Warning:** If you are using Python 2, torch.save does NOT support StringIO.StringIO as a valid file-like object. This is because the write method should return the number of bytes written; StringIO.write() does not do this.
>
> Please use something like io.BytesIO instead.

Example

```
>>> # Save to file
>>> x = torch.tensor([0, 1, 2, 3, 4])
>>> torch.save(x, 'tensor.pt')
>>> # Save to io.BytesIO buffer
>>> buffer = io.BytesIO()
>>> torch.save(x, buffer)
```

torch.**load**(*f*, *map_location=None*, *pickle_module=<module 'pickle' from '/home/yang/Essential/anaconda3/lib/python3.6/pickle.py'>*, ***pickle_load_args*)

Loads an object saved with *torch.save()* from a file.

torch.load() uses Pythons unpickling facilities but treats storages, which underlie tensors, specially. They are first deserialized on the CPU and are then moved to the device they were saved from. If this fails (e.g. because the run time system doesnt have certain devices), an exception is raised. However, storages can be dynamically remapped to an alternative set of devices using the *map_location* argument.

If *map_location* is a callable, it will be called once for each serialized storage with two arguments: storage and location. The storage argument will be the initial deserialization of the storage, residing on the CPU. Each serialized storage has a location tag associated with it which identifies the device it was saved from, and this tag is the second argument passed to map_location. The builtin location tags are *cpu* for CPU tensors and *cuda:device_id* (e.g. *cuda:2*) for CUDA tensors. *map_location* should return either None or a storage. If *map_location* returns a storage, it will be used as the final deserialized object, already moved to the right device. Otherwise, $torch.load$ will fall back to the default behavior, as if *map_location* wasnt specified.

If *map_location* is a string, it should be a device tag, where all tensors should be loaded.

Otherwise, if *map_location* is a dict, it will be used to remap location tags appearing in the file (keys), to ones that specify where to put the storages (values).

User extensions can register their own location tags and tagging and deserialization methods using *register_package*.

Parameters

- **f** – a file-like object (has to implement read, readline, tell, and seek), or a string containing a file name
- **map_location** – a function, torch.device, string 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)
- **pickle_load_args** – optional keyword arguments passed over to `pickle_module.load` and `pickle_module.Unpickler`, e.g., `encoding=...`.

Note: When you call *torch.load()* on a file which contains GPU tensors, those tensors will be loaded to GPU by default. You can call *torch.load(.., map_location=cpu)* and then `load_state_dict()` to avoid GPU RAM surge when loading a model checkpoint.

Note: In Python 3, when loading files saved by Python 2, you may encounter `UnicodeDecodeError: 'ascii' codec can't decode byte 0x...`. This is caused by the difference of handling in byte strings in Python2 and Python 3. You may use extra `encoding` keyword argument to specify how these objects should be loaded, e.g., `encoding='latin1'` decodes them to strings using `latin1` encoding, and `encoding='bytes'` keeps them as byte arrays which can be decoded later with `byte_array.decode(...)`.

Example

```
>>> torch.load('tensors.pt')
# Load all tensors onto the CPU
>>> torch.load('tensors.pt', map_location=torch.device('cpu'))
# Load all tensors onto the CPU, using a function
>>> torch.load('tensors.pt', map_location=lambda storage, loc: storage)
# Load all tensors onto GPU 1
>>> torch.load('tensors.pt', map_location=lambda storage, loc: storage.cuda(1))
# Map tensors from GPU 1 to GPU 0
>>> torch.load('tensors.pt', map_location={'cuda:1':'cuda:0'})
# Load tensor from io.BytesIO object
>>> with open('tensor.pt', 'rb') as f:
        buffer = io.BytesIO(f.read())
>>> torch.load(buffer)
```

15.5 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

15.6 Locally disabling gradient computation

The context managers `torch.no_grad()`, `torch.enable_grad()`, and `torch.set_grad_enabled()` are helpful for locally disabling and enabling gradient computation. See *Locally disabling gradient computation* for more details on their usage. These context managers are thread local, so they wont work if you send work to another thread using the `threading` module, etc.

Examples:

```
>>> x = torch.zeros(1, requires_grad=True)
>>> with torch.no_grad():
...     y = x * 2
>>> y.requires_grad
False

>>> is_train = False
>>> with torch.set_grad_enabled(is_train):
...     y = x * 2
>>> y.requires_grad
False

>>> torch.set_grad_enabled(True)  # this can also be used as a function
>>> y = x * 2
>>> y.requires_grad
True

>>> torch.set_grad_enabled(False)
>>> y = x * 2
```

(continues on next page)

(continued from previous page)

```
>>> y.requires_grad
False
```

15.7 Math operations

15.7.1 Pointwise Ops

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

Computes the element-wise absolute value of the given `input` tensor.

$$\text{out}_i = |\text{input}_i|$$

Parameters

- **input** (Tensor) – the input tensor
- **out** (Tensor, *optional*) – the output tensor

Example:

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

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

Returns a new tensor with the arccosine of the elements of `input`.

$$\text{out}_i = \cos^{-1}(\text{input}_i)$$

Parameters

- **input** (Tensor) – the input tensor
- **out** (Tensor, *optional*) – the output tensor

Example:

```
>>> a = torch.randn(4)
>>> a
tensor([ 0.3348, -0.5889,  0.2005, -0.1584])
>>> torch.acos(a)
tensor([ 1.2294,  2.2004,  1.3690,  1.7298])
```

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.

$$\text{out} = \text{input} + \text{value}$$

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

Parameters

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

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

Keyword Arguments **out** (`Tensor`, *optional*) – the output tensor

Example:

```
>>> a = torch.randn(4)
>>> a
tensor([ 0.0202,  1.0985,  1.3506, -0.6056])
>>> torch.add(a, 20)
tensor([ 20.0202,  21.0985,  21.3506,  19.3944])
```

`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` must be *broadcastable*.

$$\text{out} = \text{input} + \text{value} \times \text{other}$$

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

Parameters

- **input** (`Tensor`) – the first input tensor
- **value** (*Number*) – the scalar multiplier for `other`
- **other** (`Tensor`) – the second input tensor

Keyword Arguments **out** (`Tensor`, *optional*) – the output tensor

Example:

```
>>> a = torch.randn(4)
>>> a
tensor([-0.9732, -0.3497,  0.6245,  0.4022])
>>> b = torch.randn(4, 1)
>>> b
tensor([[ 0.3743],
        [-1.7724],
        [-0.5811],
        [-0.8017]])
>>> torch.add(a, 10, b)
tensor([[  2.7695,   3.3930,   4.3672,   4.1450],
        [-18.6971, -18.0736, -17.0994, -17.3216],
        [ -6.7845,  -6.1610,  -5.1868,  -5.4090],
        [ -8.9902,  -8.3667,  -7.3925,  -7.6147]])
```

`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*.

$$\text{out}_i = \text{tensor}_i + \text{value} \times \frac{\text{tensor1}_i}{\text{tensor2}_i}$$

The shapes of *tensor*, `tensor1`, and `tensor2` must be *broadcastable*.

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

Parameters

- **tensor** (Tensor) – the tensor to be added
- **value** (*Number, optional*) – multiplier for tensor1/tensor2
- **tensor1** (Tensor) – the numerator tensor
- **tensor2** (Tensor) – the denominator tensor
- **out** (Tensor, *optional*) – the output tensor

Example:

```
>>> t = torch.randn(1, 3)
>>> t1 = torch.randn(3, 1)
>>> t2 = torch.randn(1, 3)
>>> torch.addcdiv(t, 0.1, t1, t2)
tensor([[-0.2312, -3.6496,  0.1312],
        [-1.0428,  3.4292, -0.1030],
        [-0.5369, -0.9829,  0.0430]])
```

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*.

$$\text{out}_i = \text{tensor}_i + \text{value} \times \text{tensor1}_i \times \text{tensor2}_i$$

The shapes of *tensor*, tensor1, and tensor2 must be *broadcastable*.

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

Parameters

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

Example:

```
>>> t = torch.randn(1, 3)
>>> t1 = torch.randn(3, 1)
>>> t2 = torch.randn(1, 3)
>>> torch.addcmul(t, 0.1, t1, t2)
tensor([[-0.8635, -0.6391,  1.6174],
        [-0.7617, -0.5879,  1.7388],
        [-0.8353, -0.6249,  1.6511]])
```

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

Returns a new tensor with the arcsine of the elements of input.

$$\text{out}_i = \sin^{-1}(\text{input}_i)$$

Parameters

- **input** (Tensor) – the input tensor
- **out** (Tensor, *optional*) – the output tensor

Example:

```
>>> a = torch.randn(4)
>>> a
tensor([-0.5962,  1.4985, -0.4396,  1.4525])
>>> torch.asin(a)
tensor([-0.6387,     nan, -0.4552,     nan])
```

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

Returns a new tensor with the arctangent of the elements of `input`.

$$\text{out}_i = \tan^{-1}(\text{input}_i)$$

Parameters

- **input** (Tensor) – the input tensor
- **out** (Tensor, *optional*) – the output tensor

Example:

```
>>> a = torch.randn(4)
>>> a
tensor([ 0.2341,  0.2539, -0.6256, -0.6448])
>>> torch.atan(a)
tensor([ 0.2299,  0.2487, -0.5591, -0.5727])
```

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

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

The shapes of `input1` and `input2` must be *broadcastable*.

Parameters

- **input1** (Tensor) – the first input tensor
- **input2** (Tensor) – the second input tensor
- **out** (Tensor, *optional*) – the output tensor

Example:

```
>>> a = torch.randn(4)
>>> a
tensor([ 0.9041,  0.0196, -0.3108, -2.4423])
>>> torch.atan2(a, torch.randn(4))
tensor([ 0.9833,  0.0811, -1.9743, -1.4151])
```

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.

$$\text{out}_i = \lceil \text{input}_i \rceil = \lfloor \text{input}_i \rfloor + 1$$

Parameters

- **input** (Tensor) – the input tensor
- **out** (Tensor, *optional*) – the output tensor

Example:

```
>>> a = torch.randn(4)
>>> a
tensor([-0.6341, -1.4208, -1.0900,  0.5826])
>>> torch.ceil(a)
tensor([-0., -1., -1.,  1.])
```

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

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

$$y_i = \begin{cases} \min & \text{if } x_i < \min \\ x_i & \text{if } \min \leq x_i \leq \max \\ \max & \text{if } x_i > \max \end{cases}$$

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

Parameters

- **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 output tensor

Example:

```
>>> a = torch.randn(4)
>>> a
tensor([-1.7120,  0.1734, -0.0478, -0.0922])
>>> torch.clamp(a, min=-0.5, max=0.5)
tensor([-0.5000,  0.1734, -0.0478, -0.0922])
```

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.

Parameters

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

Example:

```
>>> a = torch.randn(4)
>>> a
tensor([-0.0299, -2.3184,  2.1593, -0.8883])
>>> torch.clamp(a, min=0.5)
tensor([ 0.5000,  0.5000,  2.1593,  0.5000])
```

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.

Parameters

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

Example:

```
>>> a = torch.randn(4)
>>> a
tensor([ 0.7753, -0.4702, -0.4599,  1.1899])
>>> torch.clamp(a, max=0.5)
tensor([ 0.5000, -0.4702, -0.4599,  0.5000])
```

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

Returns a new tensor with the cosine of the elements of `input`.

$$\text{out}_i = \cos(\text{input}_i)$$

Parameters

- **input** (Tensor) – the input tensor
- **out** (Tensor, *optional*) – the output tensor

Example:

```
>>> a = torch.randn(4)
>>> a
tensor([ 1.4309,  1.2706, -0.8562,  0.9796])
>>> torch.cos(a)
tensor([ 0.1395,  0.2957,  0.6553,  0.5574])
```

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

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

$$\text{out}_i = \cosh(\text{input}_i)$$

Parameters

- **input** (Tensor) – the input tensor
- **out** (Tensor, *optional*) – the output tensor

Example:

```
>>> a = torch.randn(4)
>>> a
tensor([ 0.1632,  1.1835, -0.6979, -0.7325])
>>> torch.cosh(a)
tensor([ 1.0133,  1.7860,  1.2536,  1.2805])
```

torch.**div**()

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

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

$$\text{out}_i = \frac{\text{input}_i}{\text{value}}$$

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

Parameters

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

Example:

```
>>> a = torch.randn(5)
>>> a
tensor([ 0.3810,  1.2774, -0.2972, -0.3719,  0.4637])
>>> torch.div(a, 0.5)
tensor([ 0.7620,  2.5548, -0.5944, -0.7439,  0.9275])
```

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

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` must be *broadcastable*.

$$\text{out}_i = \frac{\text{input}_i}{\text{other}_i}$$

Parameters

- **input** (Tensor) – the numerator tensor
- **other** (Tensor) – the denominator tensor
- **out** (Tensor, *optional*) – the output tensor

Example:

```
>>> a = torch.randn(4, 4)
>>> a
tensor([[-0.3711, -1.9353, -0.4605, -0.2917],
        [ 0.1815, -1.0111,  0.9805, -1.5923],
        [ 0.1062,  1.4581,  0.7759, -1.2344],
        [-0.1830, -0.0313,  1.1908, -1.4757]])
>>> b = torch.randn(4)
>>> b
tensor([ 0.8032,  0.2930, -0.8113, -0.2308])
>>> torch.div(a, b)
tensor([[-0.4620, -6.6051,  0.5676,  1.2637],
        [ 0.2260, -3.4507, -1.2086,  6.8988],
        [ 0.1322,  4.9764, -0.9564,  5.3480],
        [-0.2278, -0.1068, -1.4678,  6.3936]])
```

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

Computes the logarithmic derivative of the gamma function on *input*.

$$\psi(x) = \frac{d}{dx} \ln\left(\Gamma\left(x\right)\right) = \frac{\Gamma'(x)}{\Gamma(x)}$$

Parameters **input** (Tensor) – the tensor to compute the digamma function on

Example:

```
>>> a = torch.tensor([1, 0.5])
>>> torch.digamma(a)
tensor([-0.5772, -1.9635])
```

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

Computes the error function of each element. The error function is defined as follows:

$$\mathrm{erf}(x) = \frac{2}{\sqrt{\pi}} \int_0^x e^{-t^2} dt$$

Parameters

- **tensor** (Tensor) – the input tensor
- **out** (Tensor, *optional*) – the output tensor

Example:

```
>>> torch.erf(torch.tensor([0, -1., 10.]))
tensor([ 0.0000, -0.8427,  1.0000])
```

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

Computes the complementary error function of each element of `input`. The complementary error function is defined as follows:

$$\mathrm{erfc}(x) = 1 - \frac{2}{\sqrt{\pi}} \int_0^x e^{-t^2} dt$$

Parameters

- **tensor** (Tensor) – the input tensor
- **out** (Tensor, *optional*) – the output tensor

Example:

```
>>> torch.erfc(torch.tensor([0, -1., 10.]))
tensor([ 1.0000, 1.8427,  0.0000])
```

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

Computes the inverse error function of each element of `input`. The inverse error function is defined in the range $(-1, 1)$ as:

$$\mathrm{erfinv}(\mathrm{erf}(x)) = x$$

Parameters

- **input** (Tensor) – the input tensor
- **out** (Tensor, *optional*) – the output tensor

Example:

```
>>> torch.erfinv(torch.tensor([0, 0.5, -1.]))
tensor([ 0.0000,  0.4769,    -inf])
```

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

Returns a new tensor with the exponential of the elements of the input tensor `input`.

$$y_i = e^{x_i}$$

Parameters

- **input** (Tensor) – the input tensor
- **out** (Tensor, *optional*) – the output tensor

Example:

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

torch.**expm1**(*input*, *out=None*) → Tensor

Returns a new tensor with the exponential of the elements minus 1 of `input`.

$$y_i = e^{x_i} - 1$$

Parameters

- **input** (Tensor) – the input tensor
- **out** (Tensor, *optional*) – the output tensor

Example:

```
>>> torch.expm1(torch.tensor([0, math.log(2.)]))
tensor([ 0.,  1.])
```

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.

$$\text{out}_i = \lfloor \text{input}_i \rfloor$$

Parameters

- **input** (Tensor) – the input tensor
- **out** (Tensor, *optional*) – the output tensor

Example:

```
>>> a = torch.randn(4)
>>> a
tensor([-0.8166,  1.5308, -0.2530, -0.2091])
>>> torch.floor(a)
tensor([-1.,  1., -1., -1.])
```

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 `input`.

When `divisor` is a tensor, the shapes of `input` and `divisor` must be *broadcastable*.

Parameters

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

Example:

```
>>> torch.fmod(torch.tensor([-3., -2, -1, 1, 2, 3]), 2)
tensor([-1., -0., -1.,  1.,  0.,  1.])
>>> torch.fmod(torch.tensor([1., 2, 3, 4, 5]), 1.5)
tensor([ 1.0000,  0.5000,  0.0000,  1.0000,  0.5000])
```

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

Computes the fractional portion of each element in `input`.

$$\text{out}_i = \text{input}_i - \lfloor \text{input}_i \rfloor$$

Example:

```
>>> torch.frac(torch.tensor([1, 2.5, -3.2]))
tensor([ 0.0000,  0.5000, -0.2000])
```

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

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

$$\text{out}_i = \text{start}_i + \text{weight}_i \times (\text{end}_i - \text{start}_i)$$

The shapes of `start` and `end` must be *broadcastable*. If `weight` is a tensor, then the shapes of `start`, `end` must be *broadcastable*.

Parameters

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

Example:

```
>>> start = torch.arange(1., 5.)
>>> end = torch.empty(4).fill_(10)
>>> start
tensor([ 1.,  2.,  3.,  4.])
>>> end
tensor([ 10.,  10.,  10.,  10.])
>>> torch.lerp(start, end, 0.5)
tensor([ 5.5000,  6.0000,  6.5000,  7.0000])
>>> torch.lerp(start, end, torch.full_like(start, 0.5))
tensor([ 5.5000,  6.0000,  6.5000,  7.0000])
```

torch.**lgamma**()

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

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

$$y_i = \log_e(x_i)$$

Parameters

- **input** (Tensor) – the input tensor
- **out** (Tensor, *optional*) – the output tensor

Example:

```
>>> a = torch.randn(5)
>>> a
tensor([-0.7168, -0.5471, -0.8933, -1.4428, -0.1190])
>>> torch.log(a)
tensor([ nan,  nan,  nan,  nan,  nan])
```

torch.**log10**(*input*, *out=None*) → Tensor

Returns a new tensor with the logarithm to the base 10 of the elements of `input`.

$$y_i = \log_{10}(x_i)$$

Parameters

- **input** (Tensor) – the input tensor
- **out** (Tensor, *optional*) – the output tensor

Example:

```
>>> a = torch.rand(5)
>>> a
tensor([ 0.5224,  0.9354,  0.7257,  0.1301,  0.2251])


>>> torch.log10(a)
tensor([-0.2820, -0.0290, -0.1392, -0.8857, -0.6476])
```

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

Returns a new tensor with the natural logarithm of (1 + `input`).

$$y_i = \log_e(x_i + 1)$$

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

Parameters

- **input** (Tensor) – the input tensor
- **out** (Tensor, *optional*) – the output tensor

Example:

```
>>> a = torch.randn(5)
>>> a
tensor([-1.0090, -0.9923,  1.0249, -0.5372,  0.2492])
>>> torch.log1p(a)
tensor([    nan, -4.8653,  0.7055, -0.7705,  0.2225])
```

torch.**log2**(*input*, *out=None*) → Tensor

Returns a new tensor with the logarithm to the base 2 of the elements of `input`.

$$y_i = \log_2(x_i)$$

Parameters

- **input** (Tensor) – the input tensor
- **out** (Tensor, *optional*) – the output tensor

Example:

```
>>> a = torch.rand(5)
>>> a
tensor([ 0.8419,  0.8003,  0.9971,  0.5287,  0.0490])


>>> torch.log2(a)
tensor([-0.2483, -0.3213, -0.0042, -0.9196, -4.3504])
```

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.

$$\text{out}_i = \text{value} \times \text{input}_i$$

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

Parameters

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

Example:

```
>>> a = torch.randn(3)
>>> a
tensor([ 0.2015, -0.4255,  2.6087])
>>> torch.mul(a, 100)
tensor([  20.1494,  -42.5491,  260.8663])
```

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

Each element of the tensor `input` is multiplied by the corresponding element of the Tensor `other`. The resulting tensor is returned.

The shapes of `input` and `other` must be *broadcastable*.

$$\text{out}_i = \text{input}_i \times \text{other}_i$$

Parameters

- **input** (Tensor) – the first multiplicand tensor
- **other** (Tensor) – the second multiplicand tensor
- **out** (Tensor, *optional*) – the output tensor

Example:

```
>>> a = torch.randn(4, 1)
>>> a
tensor([[ 1.1207],
        [-0.3137],
        [ 0.0700],
        [ 0.8378]])
>>> b = torch.randn(1, 4)
>>> b
tensor([[ 0.5146,  0.1216, -0.5244,  2.2382]])
>>> torch.mul(a, b)
tensor([[ 0.5767,  0.1363, -0.5877,  2.5083],
        [-0.1614, -0.0382,  0.1645, -0.7021],
        [ 0.0360,  0.0085, -0.0367,  0.1567],
        [ 0.4312,  0.1019, -0.4394,  1.8753]])
```

`torch.`**`mvlgamma`**(*input*, *p*) → Tensor

Computes the multivariate log-gamma function ([reference]) with dimension p element-wise, given by

$$\log(\Gamma_p(a)) = C + \sum_{i=1}^{p} \log\left(\Gamma\left(a - \frac{i-1}{2}\right)\right)$$

where $C = \log(\pi) \times \frac{p(p-1)}{4}$ and $\Gamma(\cdot)$ is the Gamma function.

If any of the elements are less than or equal to $\frac{p-1}{2}$, then an error is thrown.

Parameters

- **input** (Tensor) – the tensor to compute the multivariate log-gamma function
- **p** (*int*) – the number of dimensions

Example:

```
>>> a = torch.empty(2, 3).uniform_(1, 2)
>>> a
tensor([[1.6835, 1.8474, 1.1929],
        [1.0475, 1.7162, 1.4180]])
>>> torch.mvlgamma(a, 2)
tensor([[0.3928, 0.4007, 0.7586],
        [1.0311, 0.3901, 0.5049]])
```

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

Returns a new tensor with the negative of the elements of `input`.

$$\text{out} = -1 \times \text{input}$$

Parameters

- **input** (Tensor) – the input tensor
- **out** (Tensor, *optional*) – the output tensor

Example:

```
>>> a = torch.randn(5)
>>> a
tensor([ 0.0090, -0.2262, -0.0682, -0.2866,  0.3940])
>>> torch.neg(a)
tensor([-0.0090,  0.2262,  0.0682,  0.2866, -0.3940])
```

torch.**polygamma**()

torch.**pow**()

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

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:

$$\text{out}_i = x_i^{\text{exponent}}$$

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

$$\text{out}_i = x_i^{\text{exponent}_i}$$

When `exponent` is a tensor, the shapes of `input` and `exponent` must be *broadcastable*.

Parameters

- **input** (Tensor) – the input tensor
- **exponent** (*float or tensor*) – the exponent value
- **out** (Tensor, *optional*) – the output tensor

Example:

```
>>> a = torch.randn(4)
>>> a
tensor([ 0.4331,  1.2475,  0.6834, -0.2791])
>>> torch.pow(a, 2)
tensor([ 0.1875,  1.5561,  0.4670,  0.0779])
>>> exp = torch.arange(1., 5.)

>>> a = torch.arange(1., 5.)
>>> a
tensor([ 1.,  2.,  3.,  4.])
>>> exp
tensor([ 1.,  2.,  3.,  4.])
>>> torch.pow(a, exp)
tensor([   1.,    4.,   27.,  256.])
```

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

`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:

$$out_i = base^{input_i}$$

Parameters

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

Example:

```
>>> exp = torch.arange(1., 5.)
>>> base = 2
>>> torch.pow(base, exp)
tensor([  2.,   4.,   8.,  16.])
```

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

Returns a new tensor with the reciprocal of the elements of `input`

$$\text{out}_i = \frac{1}{\text{input}_i}$$

Parameters

- **input** (Tensor) – the input tensor
- **out** (Tensor, *optional*) – the output tensor

Example:

```
>>> a = torch.randn(4)
>>> a
tensor([-0.4595, -2.1219, -1.4314,  0.7298])
>>> torch.reciprocal(a)
tensor([-2.1763, -0.4713, -0.6986,  1.3702])
```

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.

When `divisor` is a tensor, the shapes of `input` and `divisor` must be *broadcastable*.

Parameters

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

Example:

```
>>> torch.remainder(torch.tensor([-3., -2, -1, 1, 2, 3]), 2)
tensor([ 1.,  0.,  1.,  1.,  0.,  1.])
>>> torch.remainder(torch.tensor([1., 2, 3, 4, 5]), 1.5)
tensor([ 1.0000,  0.5000,  0.0000,  1.0000,  0.5000])
```

See also:

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.

Parameters

- **input** (Tensor) – the input tensor
- **out** (Tensor, *optional*) – the output tensor

Example:

```
>>> a = torch.randn(4)
>>> a
tensor([ 0.9920,  0.6077,  0.9734, -1.0362])
>>> torch.round(a)
tensor([ 1.,  1.,  1., -1.])
```

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

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

$$\text{out}_i = \frac{1}{\sqrt{\text{input}_i}}$$

Parameters

- **input** (Tensor) – the input tensor
- **out** (Tensor, *optional*) – the output tensor

Example:

```
>>> a = torch.randn(4)
>>> a
tensor([-0.0370,  0.2970,  1.5420, -0.9105])
>>> torch.rsqrt(a)
tensor([    nan,  1.8351,  0.8053,     nan])
```

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

Returns a new tensor with the sigmoid of the elements of `input`.

$$\text{out}_i = \frac{1}{1 + e^{-\text{input}_i}}$$

Parameters

- **input** (Tensor) – the input tensor
- **out** (Tensor, *optional*) – the output tensor

Example:

```
>>> a = torch.randn(4)
>>> a
tensor([ 0.9213,  1.0887, -0.8858, -1.7683])
>>> torch.sigmoid(a)
tensor([ 0.7153,  0.7481,  0.2920,  0.1458])
```

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

Returns a new tensor with the sign of the elements of `input`.

Parameters

- **input** (Tensor) – the input tensor
- **out** (Tensor, *optional*) – the output tensor

Example:

```
>>> a = torch.tensor([0.7, -1.2, 0., 2.3])
>>> a
tensor([ 0.7000, -1.2000,  0.0000,  2.3000])
>>> torch.sign(a)
tensor([ 1., -1.,  0.,  1.])
```

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

Returns a new tensor with the sine of the elements of `input`.

$$\text{out}_i = \sin(\text{input}_i)$$

Parameters

- **input** (Tensor) – the input tensor
- **out** (Tensor, *optional*) – the output tensor

Example:

```
>>> a = torch.randn(4)
>>> a
tensor([-0.5461,  0.1347, -2.7266, -0.2746])
>>> torch.sin(a)
tensor([-0.5194,  0.1343, -0.4032, -0.2711])
```

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

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

$$\text{out}_i = \sinh(\text{input}_i)$$

Parameters

- **input** (Tensor) – the input tensor
- **out** (Tensor, *optional*) – the output tensor

Example:

```
>>> a = torch.randn(4)
>>> a
tensor([ 0.5380, -0.8632, -0.1265,  0.9399])
>>> torch.sinh(a)
tensor([ 0.5644, -0.9744, -0.1268,  1.0845])
```

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

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

$$\text{out}_i = \sqrt{\text{input}_i}$$

Parameters

- **input** (Tensor) – the input tensor
- **out** (Tensor, *optional*) – the output tensor

Example:

```
>>> a = torch.randn(4)
>>> a
tensor([-2.0755,  1.0226,  0.0831,  0.4806])
>>> torch.sqrt(a)
tensor([    nan,  1.0112,  0.2883,  0.6933])
```

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

Returns a new tensor with the tangent of the elements of `input`.

$$\text{out}_i = \tan(\text{input}_i)$$

Parameters

- **input** (Tensor) – the input tensor
- **out** (Tensor, *optional*) – the output tensor

Example:

```
>>> a = torch.randn(4)
>>> a
tensor([-1.2027, -1.7687,  0.4412, -1.3856])
>>> torch.tan(a)
tensor([-2.5930,  4.9859,  0.4722, -5.3366])
```

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

Returns a new tensor with the hyperbolic tangent of the elements of input.

$$\text{out}_i = \tanh(\text{input}_i)$$

Parameters

- **input** (Tensor) – the input tensor
- **out** (Tensor, *optional*) – the output tensor

Example:

```
>>> a = torch.randn(4)
>>> a
tensor([ 0.8986, -0.7279,  1.1745,  0.2611])
>>> torch.tanh(a)
tensor([ 0.7156, -0.6218,  0.8257,  0.2553])
```

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

Returns a new tensor with the truncated integer values of the elements of input.

Parameters

- **input** (Tensor) – the input tensor
- **out** (Tensor, *optional*) – the output tensor

Example:

```
>>> a = torch.randn(4)
>>> a
tensor([ 3.4742,  0.5466, -0.8008, -0.9079])
>>> torch.trunc(a)
tensor([ 3.,  0., -0., -0.])
```

15.7.2 Reduction Ops

torch.**argmax**()

torch.**argmax**(*input*) → LongTensor

Returns the indices of all elements in the input tensor.

This is the second value returned by *torch.max()*. See its documentation for the exact semantics of this method.

Parameters **input** (Tensor) – the input tensor

Example:

```
>>> a = torch.randn(4, 4)
>>> a
tensor([[ 1.3398,  0.2663, -0.2686,  0.2450],
        [-0.7401, -0.8805, -0.3402, -1.1936],
        [ 0.4907, -1.3948, -1.0691, -0.3132],
        [-1.6092,  0.5419, -0.2993,  0.3195]])
>>> torch.argmax(a)
tensor(0)
```

torch.**argmax**(*input*, *dim*, *keepdim=False*) → LongTensor

Returns the indices of the maximum values of a tensor across a dimension.

This is the second value returned by *torch.max()*. See its documentation for the exact semantics of this method.

Parameters

- **input** (Tensor) – the input tensor
- **dim** (*int*) – the dimension to reduce. If None, the argmax of the flattened input is returned.
- **keepdim** (*bool*) – whether the output tensors have dim retained or not. Ignored if dim=None.

Example:

```
>>> a = torch.randn(4, 4)
>>> a
tensor([[ 1.3398,  0.2663, -0.2686,  0.2450],
        [-0.7401, -0.8805, -0.3402, -1.1936],
        [ 0.4907, -1.3948, -1.0691, -0.3132],
        [-1.6092,  0.5419, -0.2993,  0.3195]])
>>> torch.argmax(a, dim=1)
tensor([ 0,  2,  0,  1])
```

torch.**argmin**()

torch.**argmin**(*input*) → LongTensor

Returns the indices of the minimum value of all elements in the input tensor.

This is the second value returned by *torch.min()*. See its documentation for the exact semantics of this method.

Parameters **input** (Tensor) – the input tensor

Example:

```
>>> a = torch.randn(4, 4)
>>> a
tensor([[ 0.1139,  0.2254, -0.1381,  0.3687],
        [ 1.0100, -1.1975, -0.0102, -0.4732],
        [-0.9240,  0.1207, -0.7506, -1.0213],
        [ 1.7809, -1.2960,  0.9384,  0.1438]])
>>> torch.argmin(a)
tensor(13)
```

torch.**argmin**(*input*, *dim*, *keepdim=False*, *out=None*) → LongTensor

Returns the indices of the minimum values of a tensor across a dimension.

This is the second value returned by *torch.min()*. See its documentation for the exact semantics of this method.

Parameters

- **input** (Tensor) – the input tensor
- **dim** (*int*) – the dimension to reduce. If None, the argmin of the flattened input is returned.
- **keepdim** (*bool*) – whether the output tensors have dim retained or not. Ignored if dim=None.

Example:

```
>>> a = torch.randn(4, 4)
>>> a
tensor([[ 0.1139,  0.2254, -0.1381,  0.3687],
        [ 1.0100, -1.1975, -0.0102, -0.4732],
        [-0.9240,  0.1207, -0.7506, -1.0213],
        [ 1.7809, -1.2960,  0.9384,  0.1438]])
>>> torch.argmin(a, dim=1)
tensor([ 2,  1,  3,  1])
```

torch.**cumprod**(*input*, *dim*, *dtype=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.

$$y_i = x_1 \times x_2 \times x_3 \times \cdots \times x_i$$

Parameters

- **input** (Tensor) – the input tensor
- **dim** (*int*) – the dimension to do the operation over
- **dtype** (*torch.dtype*, optional) – the desired data type of returned tensor. If specified, the input tensor is casted to dtype before the operation is performed. This is useful for preventing data type overflows. Default: None.

Example:

```
>>> a = torch.randn(10)
>>> a
tensor([ 0.6001,  0.2069, -0.1919,  0.9792,  0.6727,  1.0062,  0.4126,
        -0.2129, -0.4206,  0.1968])
>>> torch.cumprod(a, dim=0)
tensor([ 0.6001,  0.1241, -0.0238, -0.0233, -0.0157, -0.0158, -0.0065,
         0.0014, -0.0006, -0.0001])

>>> a[5] = 0.0
>>> torch.cumprod(a, dim=0)
tensor([ 0.6001,  0.1241, -0.0238, -0.0233, -0.0157, -0.0000, -0.0000,
         0.0000, -0.0000, -0.0000])
```

torch.**cumsum**(*input*, *dim*, *out=None*, *dtype=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.

$$y_i = x_1 + x_2 + x_3 + \cdots + x_i$$

Parameters

- **input** (Tensor) – the input tensor
- **dim** (*int*) – the dimension to do the operation over
- **dtype** (*torch.dtype*, optional) – the desired data type of returned tensor. If specified, the input tensor is casted to `dtype` before the operation is performed. This is useful for preventing data type overflows. Default: None.

Example:

```
>>> a = torch.randn(10)
>>> a
tensor([-0.8286, -0.4890,  0.5155,  0.8443,  0.1865, -0.1752, -2.0595,
         0.1850, -1.1571, -0.4243])
>>> torch.cumsum(a, dim=0)
tensor([-0.8286, -1.3175, -0.8020,  0.0423,  0.2289,  0.0537, -2.0058,
        -1.8209, -2.9780, -3.4022])
```

`torch.`**`dist`**(*input*, *other*, *p=2*) → Tensor

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

The shapes of `input` and `other` must be *broadcastable*.

Parameters

- **input** (Tensor) – the input tensor
- **other** (Tensor) – the Right-hand-side input tensor
- **p** (*float, optional*) – the norm to be computed

Example:

```
>>> x = torch.randn(4)
>>> x
tensor([-1.5393, -0.8675,  0.5916,  1.6321])
>>> y = torch.randn(4)
>>> y
tensor([ 0.0967, -1.0511,  0.6295,  0.8360])
>>> torch.dist(x, y, 3.5)
tensor(1.6727)
>>> torch.dist(x, y, 3)
tensor(1.6973)
>>> torch.dist(x, y, 0)
tensor(inf)
>>> torch.dist(x, y, 1)
tensor(2.6537)
```

`torch.`**`logsumexp`**(*input*, *dim*, *keepdim=False*, *out=None*)

Returns the log of summed exponentials of each row of the `input` tensor in the given dimension `dim`. The computation is numerically stabilized.

For summation index j given by *dim* and other indices i, the result is

$$\text{logsumexp}(x)_i = \log \sum_j \exp(x_{ij})$$

If keepdim is True, the output tensor is of the same size as input except in the dimension(s) dim where it is of size 1. Otherwise, dim is squeezed (see *torch.squeeze()*), resulting in the output tensor having 1 (or len(dim)) fewer dimension(s).

Parameters

- **input** (Tensor) – the input tensor
- **dim** (*int or tuple of python:ints*) – the dimension or dimensions to reduce
- **keepdim** (*bool*) – whether the output tensor has dim retained or not
- **out** (Tensor, *optional*) – the output tensor

Example::

```
>>> a = torch.randn(3, 3)
>>> torch.logsumexp(a, 1)
tensor([ 0.8442,  1.4322,  0.8711])
```

torch.**mean**()

torch.**mean**(*input*) → Tensor

Returns the mean value of all elements in the input tensor.

Parameters **input** (Tensor) – the input tensor

Example:

```
>>> a = torch.randn(1, 3)
>>> a
tensor([[ 0.2294, -0.5481,  1.3288]])
>>> torch.mean(a)
tensor(0.3367)
```

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

Returns the mean value of each row of the input tensor in the given dimension dim. If dim is a list of dimensions, reduce over all of them.

If keepdim is True, the output tensor is of the same size as input except in the dimension(s) dim where it is of size 1. Otherwise, dim is squeezed (see *torch.squeeze()*), resulting in the output tensor having 1 (or len(dim)) fewer dimension(s).

Parameters

- **input** (Tensor) – the input tensor
- **dim** (*int or tuple of python:ints*) – the dimension or dimensions to reduce
- **keepdim** (*bool*) – whether the output tensor has dim retained or not
- **out** (Tensor) – the output tensor

Example:

```
>>> a = torch.randn(4, 4)
>>> a
tensor([[-0.3841,  0.6320,  0.4254, -0.7384],
        [-0.9644,  1.0131, -0.6549, -1.4279],
        [-0.2951, -1.3350, -0.7694,  0.5600],
```

(continues on next page)

(continued from previous page)

```
        [ 1.0842, -0.9580,  0.3623,  0.2343]])
>>> torch.mean(a, 1)
tensor([-0.0163, -0.5085, -0.4599,  0.1807])
>>> torch.mean(a, 1, True)
tensor([[-0.0163],
        [-0.5085],
        [-0.4599],
        [ 0.1807]])
```

torch.**median**()

torch.**median**(*input*) → Tensor

Returns the median value of all elements in the `input` tensor.

Parameters **input** (Tensor) – the input tensor

Example:

```
>>> a = torch.randn(1, 3)
>>> a
tensor([[ 1.5219, -1.5212,  0.2202]])
>>> torch.median(a)
tensor(0.2202)
```

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

Returns a namedtuple `(values, indices)` where `values` is the median value of each row of the `input` tensor in the given dimension `dim`. And `indices` is the index location of each median value found.

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

If `keepdim` is `True`, the output tensors are of the same size as `input` except in the dimension `dim` where they are of size 1. Otherwise, `dim` is squeezed (see *`torch.squeeze()`*), resulting in the outputs tensor having 1 fewer dimension than `input`.

Parameters

- **input** (Tensor) – the input tensor
- **dim** (*int*) – the dimension to reduce
- **keepdim** (*bool*) – whether the output tensors have `dim` retained or not
- **values** (Tensor, *optional*) – the output tensor
- **indices** (Tensor, *optional*) – the output index tensor

Example:

```
>>> a = torch.randn(4, 5)
>>> a
tensor([[ 0.2505, -0.3982, -0.9948,  0.3518, -1.3131],
        [ 0.3180, -0.6993,  1.0436,  0.0438,  0.2270],
        [-0.2751,  0.7303,  0.2192,  0.3321,  0.2488],
        [ 1.0778, -1.9510,  0.7048,  0.4742, -0.7125]])
>>> torch.median(a, 1)
torch.return_types.median(values=tensor([-0.3982,  0.2270,  0.2488,  0.4742]),␣
↪indices=tensor([1, 4, 4, 3]))
```

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

Returns a namedtuple `(values, indices)` where `values` is the mode value of each row of the `input` tensor in the given dimension `dim`, i.e. a value which appears most often in that row, and `indices` is the index location of each mode value found.

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

If `keepdim` is `True`, the output tensors are of the same size as `input` except in the dimension `dim` where they are of size 1. Otherwise, `dim` is squeezed (see *torch.squeeze()*), resulting in the output tensors having 1 fewer dimension than `input`.

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

Parameters

- **input** (Tensor) – the input tensor
- **dim** (*int*) – the dimension to reduce
- **keepdim** (*bool*) – whether the output tensors have `dim` retained or not
- **values** (Tensor, *optional*) – the output tensor
- **indices** (Tensor, *optional*) – the output index tensor

Example:

```
>>> a = torch.randint(10, (5,))
>>> a
tensor([6, 5, 1, 0, 2])
>>> b = a + (torch.randn(50, 1) * 5).long()
>>> torch.mode(b, 0)
torch.return_types.mode(values=tensor([6, 5, 1, 0, 2]), indices=tensor([2, 2, 2,␣
↪2, 2]))
```

torch.**norm**(*input*, *p='fro'*, *dim=None*, *keepdim=False*, *out=None*, *dtype=None*)

Returns the matrix norm or vector norm of a given tensor.

Parameters

- **input** (Tensor) – the input tensor
- **p** (*int*, *float*, *inf*, *-inf*, *'fro'*, *'nuc'*, *optional*) – the order of norm. Default: `'fro'` The following norms can be calculated:

ord	matrix norm	vector norm
None	Frobenius norm	2-norm
fro	Frobenius norm	–
nuc	nuclear norm	–
Other	as vec norm when dim is None	sum(abs(x)**ord)**(1./ord)

- **dim** (*int*, *2-tuple of python:ints*, *2-list of python:ints*, *optional*) – If it is an int, vector norm will be calculated, if it is 2-tuple of ints, matrix norm will be calculated. If the value is None, matrix norm will be calculated when the input tensor only has two dimensions, vector norm will be calculated when the input tensor only has one dimension. If the input tensor has more than two dimensions, the vector norm will be applied to last dimension.

- **keepdim** (*bool, optional*) – whether the output tensors have `dim` retained or not. Ignored if `dim` = `None` and `out` = `None`. Default: `False`
- **out** (`Tensor`, *optional*) – the output tensor. Ignored if `dim` = `None` and `out` = `None`.
- **dtype** (*torch.dtype*, optional) – the desired data type of returned tensor. If specified, the input tensor is casted to :attr:dtype while performing the operation. Default: None.

Example:

```
>>> import torch
>>> a = torch.arange(9, dtype= torch.float) - 4
>>> b = a.reshape((3, 3))
>>> torch.norm(a)
tensor(7.7460)
>>> torch.norm(b)
tensor(7.7460)
>>> torch.norm(a, float('inf'))
tensor(4.)
>>> torch.norm(b, float('inf'))
tensor(4.)
>>> c = torch.tensor([[ 1, 2, 3],[-1, 1, 4]] , dtype= torch.float)
>>> torch.norm(c, dim=0)
tensor([1.4142, 2.2361, 5.0000])
>>> torch.norm(c, dim=1)
tensor([3.7417, 4.2426])
>>> torch.norm(c, p=1, dim=1)
tensor([6., 6.])
>>> d = torch.arange(8, dtype= torch.float).reshape(2,2,2)
>>> torch.norm(d, dim=(1,2))
tensor([ 3.7417, 11.2250])
>>> torch.norm(d[0, :, :]), torch.norm(d[1, :, :])
(tensor(3.7417), tensor(11.2250))
```

`torch.`**`prod`**`()`

`torch.`**`prod`**(*input*, *dtype=None*) → Tensor

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

Parameters

- **input** (`Tensor`) – the input tensor
- **dtype** (*torch.dtype*, optional) – the desired data type of returned tensor. If specified, the input tensor is casted to `dtype` before the operation is performed. This is useful for preventing data type overflows. Default: None.

Example:

```
>>> a = torch.randn(1, 3)
>>> a
tensor([[-0.8020,  0.5428, -1.5854]])
>>> torch.prod(a)
tensor(0.6902)
```

`torch.`**`prod`**(*input*, *dim*, *keepdim=False*, *dtype=None*) → Tensor

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

If keepdim is True, the output tensor is of the same size as input except in the dimension dim where it is of size 1. Otherwise, dim is squeezed (see *torch.squeeze()*), resulting in the output tensor having 1 fewer dimension than input.

Parameters

- **input** (Tensor) – the input tensor
- **dim** (*int*) – the dimension to reduce
- **keepdim** (*bool*) – whether the output tensor has dim retained or not
- **dtype** (*torch.dtype*, optional) – the desired data type of returned tensor. If specified, the input tensor is casted to dtype before the operation is performed. This is useful for preventing data type overflows. Default: None.

Example:

```
>>> a = torch.randn(4, 2)
>>> a
tensor([[ 0.5261, -0.3837],
        [ 1.1857, -0.2498],
        [-1.1646,  0.0705],
        [ 1.1131, -1.0629]])
>>> torch.prod(a, 1)
tensor([-0.2018, -0.2962, -0.0821, -1.1831])
```

torch.**std**()

torch.**std**(*input*, *unbiased=True*) → Tensor

Returns the standard-deviation of all elements in the input tensor.

If unbiased is False, then the standard-deviation will be calculated via the biased estimator. Otherwise, Bessels correction will be used.

Parameters

- **input** (Tensor) – the input tensor
- **unbiased** (*bool*) – whether to use the unbiased estimation or not

Example:

```
>>> a = torch.randn(1, 3)
>>> a
tensor([[-0.8166, -1.3802, -0.3560]])
>>> torch.std(a)
tensor(0.5130)
```

torch.**std**(*input*, *dim*, *keepdim=False*, *unbiased=True*, *out=None*) → Tensor

Returns the standard-deviation of each row of the input tensor in the dimension dim. If dim is a list of dimensions, reduce over all of them.

If keepdim is True, the output tensor is of the same size as input except in the dimension(s) dim where it is of size 1. Otherwise, dim is squeezed (see *torch.squeeze()*), resulting in the output tensor having 1 (or len(dim)) fewer dimension(s).

If unbiased is False, then the standard-deviation will be calculated via the biased estimator. Otherwise, Bessels correction will be used.

Parameters

- **input** (Tensor) – the input tensor
- **dim** (*int or tuple of python:ints*) – the dimension or dimensions to reduce
- **keepdim** (*bool*) – whether the output tensor has dim retained or not
- **unbiased** (*bool*) – whether to use the unbiased estimation or not
- **out** (Tensor, *optional*) – the output tensor

Example:

```
>>> a = torch.randn(4, 4)
>>> a
tensor([[ 0.2035,  1.2959,  1.8101, -0.4644],
        [ 1.5027, -0.3270,  0.5905,  0.6538],
        [-1.5745,  1.3330, -0.5596, -0.6548],
        [ 0.1264, -0.5080,  1.6420,  0.1992]])
>>> torch.std(a, dim=1)
tensor([ 1.0311,  0.7477,  1.2204,  0.9087])
```

torch.**sum**()

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

Returns the sum of all elements in the input tensor.

Parameters

- **input** (Tensor) – the input tensor
- **dtype** (*torch.dtype*, optional) – the desired data type of returned tensor. If specified, the input tensor is casted to dtype before the operation is performed. This is useful for preventing data type overflows. Default: None.

Example:

```
>>> a = torch.randn(1, 3)
>>> a
tensor([[ 0.1133, -0.9567,  0.2958]])
>>> torch.sum(a)
tensor(-0.5475)
```

torch.**sum**(*input*, *dim*, *keepdim=False*, *dtype=None*) → Tensor

Returns the sum of each row of the input tensor in the given dimension dim. If dim is a list of dimensions, reduce over all of them.

If keepdim is True, the output tensor is of the same size as input except in the dimension(s) dim where it is of size 1. Otherwise, dim is squeezed (see *torch.squeeze()*), resulting in the output tensor having 1 (or len(dim)) fewer dimension(s).

Parameters

- **input** (Tensor) – the input tensor
- **dim** (*int or tuple of python:ints*) – the dimension or dimensions to reduce
- **keepdim** (*bool*) – whether the output tensor has dim retained or not
- **dtype** (*torch.dtype*, optional) – the desired data type of returned tensor. If specified, the input tensor is casted to dtype before the operation is performed. This is useful for preventing data type overflows. Default: None.

Example:

```
>>> a = torch.randn(4, 4)
>>> a
tensor([[ 0.0569, -0.2475,  0.0737, -0.3429],
        [-0.2993,  0.9138,  0.9337, -1.6864],
        [ 0.1132,  0.7892, -0.1003,  0.5688],
        [ 0.3637, -0.9906, -0.4752, -1.5197]])
>>> torch.sum(a, 1)
tensor([-0.4598, -0.1381,  1.3708, -2.6217])
>>> b = torch.arange(4 * 5 * 6).view(4, 5, 6)
>>> torch.sum(b, (2, 1))
tensor([  435.,  1335.,  2235.,  3135.])
```

torch.**unique**(*input*, *sorted=True*, *return_inverse=False*, *dim=None*)

Returns the unique scalar elements of the input tensor as a 1-D tensor.

Parameters

- **input** (Tensor) – the input tensor
- **sorted** (*bool*) – Whether to sort the unique elements in ascending order before returning as output.
- **return_inverse** (*bool*) – Whether to also return the indices for where elements in the original input ended up in the returned unique list.
- **dim** (*int*) – the dimension to apply unique. If None, the unique of the flattened input is returned. default: None

Returns

A tensor or a tuple of tensors containing

- **output** (*Tensor*): the output list of unique scalar elements.
- **inverse_indices** (*Tensor*): (optional) if return_inverse is True, there will be a 2nd returned tensor (same shape as input) representing the indices for where elements in the original input map to in the output; otherwise, this function will only return a single tensor.

Return type (*Tensor*, *Tensor* (optional))

Example:

```
>>> output = torch.unique(torch.tensor([1, 3, 2, 3], dtype=torch.long))
>>> output
tensor([ 2,  3,  1])

>>> output, inverse_indices = torch.unique(
        torch.tensor([1, 3, 2, 3], dtype=torch.long), sorted=True, return_
↪inverse=True)
>>> output
tensor([ 1,  2,  3])
>>> inverse_indices
tensor([ 0,  2,  1,  2])

>>> output, inverse_indices = torch.unique(
        torch.tensor([[1, 3], [2, 3]], dtype=torch.long), sorted=True, return_
↪inverse=True)
>>> output
tensor([ 1,  2,  3])
```

(continues on next page)

(continued from previous page)

```
>>> inverse_indices
tensor([[ 0,  2],
        [ 1,  2]])
```

torch.**var**()

torch.**var**(*input*, *unbiased=True*) → Tensor

Returns the variance of all elements in the `input` tensor.

If `unbiased` is `False`, then the variance will be calculated via the biased estimator. Otherwise, Bessels correction will be used.

Parameters

- **input** (`Tensor`) – the input tensor
- **unbiased** (*bool*) – whether to use the unbiased estimation or not

Example:

```
>>> a = torch.randn(1, 3)
>>> a
tensor([[-0.3425, -1.2636, -0.4864]])
>>> torch.var(a)
tensor(0.2455)
```

torch.**var**(*input*, *dim*, *keepdim=False*, *unbiased=True*, *out=None*) → Tensor

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

If `keepdim` is `True`, the output tensor is of the same size as `input` except in the dimension(s) `dim` where it is of size 1. Otherwise, `dim` is squeezed (see *torch.squeeze()*), resulting in the output tensor having 1 (or `len(dim)`) fewer dimension(s).

If `unbiased` is `False`, then the variance will be calculated via the biased estimator. Otherwise, Bessels correction will be used.

Parameters

- **input** (`Tensor`) – the input tensor
- **dim** (*int or tuple of python:ints*) – the dimension or dimensions to reduce
- **keepdim** (*bool*) – whether the output tensor has `dim` retained or not
- **unbiased** (*bool*) – whether to use the unbiased estimation or not
- **out** (`Tensor`, *optional*) – the output tensor

Example:

```
>>> a = torch.randn(4, 4)
>>> a
tensor([[-0.3567,  1.7385, -1.3042,  0.7423],
        [ 1.3436, -0.1015, -0.9834, -0.8438],
        [ 0.6056,  0.1089, -0.3112, -1.4085],
        [-0.7700,  0.6074, -0.1469,  0.7777]])
>>> torch.var(a, 1)
tensor([ 1.7444,  1.1363,  0.7356,  0.5112])
```

15.7.3 Comparison Ops

`torch.allclose` (*self*, *other*, *rtol=1e-05*, *atol=1e-08*, *equal_nan=False*) → bool

This function checks if all `self` and `other` satisfy the condition:

$$|\text{self} - \text{other}| \leq \texttt{atol} + \texttt{rtol} \times |\text{other}|$$

elementwise, for all elements of `self` and `other`. The behaviour of this function is analogous to numpy.allclose

Parameters

- **self** (Tensor) – first tensor to compare
- **other** (Tensor) – second tensor to compare
- **atol** (*float, optional*) – absolute tolerance. Default: 1e-08
- **rtol** (*float, optional*) – relative tolerance. Default: 1e-05
- **equal_nan** (*float, optional*) – if `True`, then two `NaN` s will be compared as equal. Default: `False`

Example:

```
>>> torch.allclose(torch.tensor([10000., 1e-07]), torch.tensor([10000.1, 1e-08]))
False
>>> torch.allclose(torch.tensor([10000., 1e-08]), torch.tensor([10000.1, 1e-09]))
True
>>> torch.allclose(torch.tensor([1.0, float('nan')]), torch.tensor([1.0, float(
↪'nan')]))
False
>>> torch.allclose(torch.tensor([1.0, float('nan')]), torch.tensor([1.0, float(
↪'nan')]), equal_nan=True)
True
```

`torch.argsort` (*input*, *dim=-1*, *descending=False*, *out=None*) → LongTensor

Returns the indices that sort a tensor along a given dimension in ascending order by value.

This is the second value returned by *torch.sort()*. See its documentation for the exact semantics of this method.

Parameters

- **input** (Tensor) – the input tensor
- **dim** (*int, optional*) – the dimension to sort along
- **descending** (*bool, optional*) – controls the sorting order (ascending or descending)

Example:

```
>>> a = torch.randn(4, 4)
>>> a
tensor([[ 0.0785,  1.5267, -0.8521,  0.4065],
        [ 0.1598,  0.0788, -0.0745, -1.2700],
        [ 1.2208,  1.0722, -0.7064,  1.2564],
        [ 0.0669, -0.2318, -0.8229, -0.9280]])


>>> torch.argsort(a, dim=1)
```

(continues on next page)

(continued from previous page)

```
tensor([[2, 0, 3, 1],
        [3, 2, 1, 0],
        [2, 1, 0, 3],
        [3, 2, 1, 0]])
```

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

Computes element-wise equality

The second argument can be a number or a tensor whose shape is *broadcastable* with the first argument.

Parameters

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

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

Return type *Tensor*

Example:

```
>>> torch.eq(torch.tensor([[1, 2], [3, 4]]), torch.tensor([[1, 1], [4, 4]]))
tensor([[ 1,  0],
        [ 0,  1]], dtype=torch.uint8)
```

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 input ≥ other element-wise.

The second argument can be a number or a tensor whose shape is *broadcastable* with the first argument.

Parameters

- **input** (Tensor) – the tensor to compare
- **other** (Tensor *or* *float*) – the tensor or value to compare
- **out** (Tensor, *optional*) – the output tensor that must be a *ByteTensor*

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

Return type *Tensor*

Example:

```
>>> torch.ge(torch.tensor([[1, 2], [3, 4]]), torch.tensor([[1, 1], [4, 4]]))
tensor([[ 1,  1],
        [ 0,  1]], dtype=torch.uint8)
```

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

Computes input > other element-wise.

The second argument can be a number or a tensor whose shape is *broadcastable* with the first argument.

Parameters

- **input** (Tensor) – the tensor to compare
- **other** (Tensor *or* *float*) – the tensor or value to compare
- **out** (Tensor, *optional*) – the output tensor that must be a *ByteTensor*

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

Return type *Tensor*

Example:

```
>>> torch.gt(torch.tensor([[1, 2], [3, 4]]), torch.tensor([[1, 1], [4, 4]]))
tensor([[ 0,  1],
        [ 0,  0]], dtype=torch.uint8)
```

`torch.`**`isfinite`**(*tensor*)

Returns a new tensor with boolean elements representing if each element is *Finite* or not.

Parameters **tensor** (Tensor) – A tensor to check

Returns A `torch.ByteTensor` containing a 1 at each location of finite elements and 0 otherwise

Return type *Tensor*

Example:

```
>>> torch.isfinite(torch.tensor([1, float('inf'), 2, float('-inf'), float('nan
↪')]))
tensor([ 1,  0,  1,  0,  0], dtype=torch.uint8)
```

`torch.`**`isinf`**(*tensor*)

Returns a new tensor with boolean elements representing if each element is *+/-INF* or not.

Parameters **tensor** (Tensor) – A tensor to check

Returns A `torch.ByteTensor` containing a 1 at each location of *+/-INF* elements and 0 otherwise

Return type *Tensor*

Example:

```
>>> torch.isinf(torch.tensor([1, float('inf'), 2, float('-inf'), float('nan')]))
tensor([ 0,  1,  0,  1,  0], dtype=torch.uint8)
```

`torch.`**`isnan`**()

Returns a new tensor with boolean elements representing if each element is *NaN* or not.

Parameters **tensor** (Tensor) – A tensor to check

Returns A `torch.ByteTensor` containing a 1 at each location of *NaN* elements.

Return type *Tensor*

Example:

```
>>> torch.isnan(torch.tensor([1, float('nan'), 2]))
tensor([ 0,  1,  0], dtype=torch.uint8)
```

`torch.`**`kthvalue`**(*input*, *k*, *dim=None*, *keepdim=False*, *out=None*) -> (*Tensor*, *LongTensor*)

Returns a namedtuple `(values, indices)` where `values` is the `k` th smallest element of each row of the `input` tensor in the given dimension `dim`. And `indices` is the index location of each element found.

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

If `keepdim` is `True`, both the `values` and `indices` tensors are the same size as `input`, except in the dimension `dim` where they are of size 1. Otherwise, `dim` is squeezed (see *`torch.squeeze()`*), resulting in both the `values` and `indices` tensors having 1 fewer dimension than the `input` tensor.

Parameters

- **input** (`Tensor`) – the input tensor
- **k** (*int*) – k for the k-th smallest element
- **dim** (*int, optional*) – the dimension to find the kth value along
- **keepdim** (*bool*) – whether the output tensors have `dim` retained or not
- **out** (*tuple, optional*) – the output tuple of (Tensor, LongTensor) can be optionally given to be used as output buffers

Example:

```
>>> x = torch.arange(1., 6.)
>>> x
tensor([ 1.,  2.,  3.,  4.,  5.])
>>> torch.kthvalue(x, 4)
torch.return_types.kthvalue(values=tensor(4.), indices=tensor(3))

>>> x=torch.arange(1.,7.).resize_(2,3)
>>> x
tensor([[ 1.,  2.,  3.],
        [ 4.,  5.,  6.]])
>>> torch.kthvalue(x, 2, 0, True)
torch.return_types.kthvalue(values=tensor([[4., 5., 6.]]), indices=tensor([[1, 1,␣
↪1]]))
```

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

Computes input $\leq$ other element-wise.

The second argument can be a number or a tensor whose shape is *broadcastable* with the first argument.

Parameters

- **input** (`Tensor`) – the tensor to compare
- **other** (`Tensor` or *float*) – the tensor or value to compare
- **out** (`Tensor`, *optional*) – the output tensor that must be a *ByteTensor*

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

Return type *Tensor*

Example:

```
>>> torch.le(torch.tensor([[1, 2], [3, 4]]), torch.tensor([[1, 1], [4, 4]]))
tensor([[ 1,  0],
        [ 1,  1]], dtype=torch.uint8)
```

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

Computes input $<$ other element-wise.

The second argument can be a number or a tensor whose shape is *broadcastable* with the first argument.

Parameters

- **input** (Tensor) – the tensor to compare
- **other** (Tensor *or* *float*) – the tensor or value to compare
- **out** (Tensor, *optional*) – the output tensor that must be a *ByteTensor*

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

Return type *Tensor*

Example:

```
>>> torch.lt(torch.tensor([[1, 2], [3, 4]]), torch.tensor([[1, 1], [4, 4]]))
tensor([[ 0,  0],
        [ 1,  0]], dtype=torch.uint8)
```

torch.**max**()

torch.**max**(*input*) → Tensor

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

Parameters **input** (Tensor) – the input tensor

Example:

```
>>> a = torch.randn(1, 3)
>>> a
tensor([[ 0.6763,  0.7445, -2.2369]])
>>> torch.max(a)
tensor(0.7445)
```

torch.**max**(*input*, *dim*, *keepdim=False*, *out=None*) -> (*Tensor*, *LongTensor*)

Returns a namedtuple `(values, indices)` where `values` is the maximum value of each row of the `input` tensor in the given dimension `dim`. And `indices` is the index location of each maximum value found (argmax).

If `keepdim` is `True`, the output tensors are of the same size as `input` except in the dimension `dim` where they are of size 1. Otherwise, `dim` is squeezed (see *torch.squeeze()*), resulting in the output tensors having 1 fewer dimension than `input`.

Parameters

- **input** (Tensor) – the input tensor
- **dim** (*int*) – the dimension to reduce
- **keepdim** (*bool*, *optional*) – whether the output tensors have `dim` retained or not. Default: `False`.
- **out** (*tuple*, *optional*) – the result tuple of two output tensors (max, max_indices)

Example:

```
>>> a = torch.randn(4, 4)
>>> a
tensor([[-1.2360, -0.2942, -0.1222,  0.8475],
        [ 1.1949, -1.1127, -2.2379, -0.6702],
        [ 1.5717, -0.9207,  0.1297, -1.8768],
        [-0.6172,  1.0036, -0.6060, -0.2432]])
>>> torch.max(a, 1)
torch.return_types.max(values=tensor([0.8475, 1.1949, 1.5717, 1.0036]),␣
↪indices=tensor([3, 0, 0, 1]))
```

(continues on next page)

(continued from previous page)

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 maximum is taken.

The shapes of `input` and `other` dont need to match, but they must be *broadcastable*.

$$\text{out}_i = \max(\text{tensor}_i, \text{other}_i)$$

Note: When the shapes do not match, the shape of the returned output tensor follows the *broadcasting rules*.

Parameters

- **input** (Tensor) – the input tensor
- **other** (Tensor) – the second input tensor
- **out** (Tensor, *optional*) – the output tensor

Example:

```
>>> a = torch.randn(4)
>>> a
tensor([ 0.2942, -0.7416,  0.2653, -0.1584])
>>> b = torch.randn(4)
>>> b
tensor([ 0.8722, -1.7421, -0.4141, -0.5055])
>>> torch.max(a, b)
tensor([ 0.8722, -0.7416,  0.2653, -0.1584])
```

torch.**min**()

torch.**min**(*input*) → Tensor

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

Parameters **input** (Tensor) – the input tensor

Example:

```
>>> a = torch.randn(1, 3)
>>> a
tensor([[ 0.6750,  1.0857,  1.7197]])
>>> torch.min(a)
tensor(0.6750)
```

torch.**min**(*input*, *dim*, *keepdim=False*, *out=None*) -> (*Tensor*, *LongTensor*)

Returns a namedtuple `(values, indices)` where `values` is the minimum value of each row of the `input` tensor in the given dimension `dim`. And `indices` is the index location of each minimum value found (argmin).

If `keepdim` is `True`, the output tensors are of the same size as `input` except in the dimension `dim` where they are of size 1. Otherwise, `dim` is squeezed (see *torch.squeeze()*), resulting in the output tensors having 1 fewer dimension than `input`.

Parameters

- **input** (Tensor) – the input tensor
- **dim** (*int*) – the dimension to reduce
- **keepdim** (*bool*) – whether the output tensors have dim retained or not
- **out** (*tuple, optional*) – the tuple of two output tensors (min, min_indices)

Example:

```
>>> a = torch.randn(4, 4)
>>> a
tensor([[-0.6248,  1.1334, -1.1899, -0.2803],
        [-1.4644, -0.2635, -0.3651,  0.6134],
        [ 0.2457,  0.0384,  1.0128,  0.7015],
        [-0.1153,  2.9849,  2.1458,  0.5788]])
>>> torch.min(a, 1)
torch.return_types.min(values=tensor([-1.1899, -1.4644,  0.0384, -0.1153]),␣
↪indices=tensor([2, 0, 1, 0]))
```

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 minimum is taken. The resulting tensor is returned.

The shapes of input and other dont need to match, but they must be *broadcastable*.

$$\text{out}_i = \min(\text{tensor}_i, \text{other}_i)$$

Note: When the shapes do not match, the shape of the returned output tensor follows the *broadcasting rules*.

Parameters

- **input** (Tensor) – the input tensor
- **other** (Tensor) – the second input tensor
- **out** (Tensor, *optional*) – the output tensor

Example:

```
>>> a = torch.randn(4)
>>> a
tensor([ 0.8137, -1.1740, -0.6460,  0.6308])
>>> b = torch.randn(4)
>>> b
tensor([-0.1369,  0.1555,  0.4019, -0.1929])
>>> torch.min(a, b)
tensor([-0.1369, -1.1740, -0.6460, -0.1929])
```

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

Computes $input \neq other$ element-wise.

The second argument can be a number or a tensor whose shape is *broadcastable* with the first argument.

Parameters

- **input** (Tensor) – the tensor to compare

- **other** (*Tensor or float*) – the tensor or value to compare
- **out** (*Tensor, optional*) – the output tensor that must be a *ByteTensor*

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

Return type *Tensor*

Example:

```
>>> torch.ne(torch.tensor([[1, 2], [3, 4]]), torch.tensor([[1, 1], [4, 4]]))
tensor([[ 0,  1],
        [ 1,  0]], dtype=torch.uint8)
```

`torch.`**`sort`**(*input*, *dim=-1*, *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 namedtuple of (values, indices) is returned, where the *values* are the sorted values and *indices* are the indices of the elements in the original *input* tensor.

Parameters

- **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*) that can be optionally given to be used as output buffers

Example:

```
>>> x = torch.randn(3, 4)
>>> sorted, indices = torch.sort(x)
>>> sorted
tensor([[-0.2162,  0.0608,  0.6719,  2.3332],
        [-0.5793,  0.0061,  0.6058,  0.9497],
        [-0.5071,  0.3343,  0.9553,  1.0960]])
>>> indices
tensor([[ 1,  0,  2,  3],
        [ 3,  1,  0,  2],
        [ 0,  3,  1,  2]])

>>> sorted, indices = torch.sort(x, 0)
>>> sorted
tensor([[-0.5071, -0.2162,  0.6719, -0.5793],
        [ 0.0608,  0.0061,  0.9497,  0.3343],
        [ 0.6058,  0.9553,  1.0960,  2.3332]])
>>> indices
tensor([[ 2,  0,  0,  1],
        [ 0,  1,  1,  2],
        [ 1,  2,  2,  0]])
```

`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 namedtuple 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

Parameters

- **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) that can be optionally given to be used as output buffers

Example:

```
>>> x = torch.arange(1., 6.)
>>> x
tensor([ 1.,  2.,  3.,  4.,  5.])
>>> torch.topk(x, 3)
torch.return_types.topk(values=tensor([5., 4., 3.]), indices=tensor([4, 3, 2]))
```

15.7.4 Spectral Ops

`torch.fft`(*input*, *signal_ndim*, *normalized=False*) → Tensor

Complex-to-complex Discrete Fourier Transform

This method computes the complex-to-complex discrete Fourier transform. Ignoring the batch dimensions, it computes the following expression:

$$X[\omega_1, \dots, \omega_d] = \sum_{n_1=0}^{N_1-1} \cdots \sum_{n_d=0}^{N_d-1} x[n_1, \dots, n_d] e^{-j\ 2\pi \sum_{i=0}^{d} \frac{\omega_i n_i}{N_i}},$$

where d = `signal_ndim` is number of dimensions for the signal, and N_i is the size of signal dimension i.

This method supports 1D, 2D and 3D complex-to-complex transforms, indicated by `signal_ndim`. `input` must be a tensor with last dimension of size 2, representing the real and imaginary components of complex numbers, and should have at least `signal_ndim + 1` dimensions with optionally arbitrary number of leading batch dimensions. If `normalized` is set to `True`, this normalizes the result by dividing it with $\sqrt{\prod_{i=1}^{K} N_i}$ so that the operator is unitary.

Returns the real and the imaginary parts together as one tensor of the same shape of `input`.

The inverse of this function is *ifft()*.

Note: For CUDA tensors, an LRU cache is used for cuFFT plans to speed up repeatedly running FFT methods on tensors of same geometry with same same configuration.

Changing `torch.backends.cuda.cufft_plan_cache.max_size` (default is 4096 on CUDA 10 and newer, and 1023 on older CUDA versions) controls the capacity of this cache. Some cuFFT plans may allocate GPU memory. You can use `torch.backends.cuda.cufft_plan_cache.size` to query the number

of plans currently in cache, and `torch.backends.cuda.cufft_plan_cache.clear()` to clear the cache.

Warning: For CPU tensors, this method is currently only available with MKL. Use `torch.backends.mkl.is_available()` to check if MKL is installed.

Parameters

- **input** (*Tensor*) – the input tensor of at least `signal_ndim + 1` dimensions
- **signal_ndim** (*int*) – the number of dimensions in each signal. `signal_ndim` can only be 1, 2 or 3
- **normalized** (*bool, optional*) – controls whether to return normalized results. Default: `False`

Returns A tensor containing the complex-to-complex Fourier transform result

Return type *Tensor*

Example:

```
>>> # unbatched 2D FFT
>>> x = torch.randn(4, 3, 2)
>>> torch.fft(x, 2)
tensor([[[-0.0876,  1.7835],
         [-2.0399, -2.9754],
         [ 4.4773, -5.0119]],

        [[-1.5716,  2.7631],
         [-3.8846,  5.2652],
         [ 0.2046, -0.7088]],

        [[ 1.9938, -0.5901],
         [ 6.5637,  6.4556],
         [ 2.9865,  4.9318]],

        [[ 7.0193,  1.1742],
         [-1.3717, -2.1084],
         [ 2.0289,  2.9357]]])
>>> # batched 1D FFT
>>> torch.fft(x, 1)
tensor([[[ 1.8385,  1.2827],
         [-0.1831,  1.6593],
         [ 2.4243,  0.5367]],

        [[-0.9176, -1.5543],
         [-3.9943, -2.9860],
         [ 1.2838, -2.9420]],

        [[-0.8854, -0.6860],
         [ 2.4450,  0.0808],
         [ 1.3076, -0.5768]],

        [[-0.1231,  2.7411],
         [-0.3075, -1.7295],
```

(continues on next page)

(continued from previous page)

```
        [-0.5384, -2.0299]]])
>>> # arbitrary number of batch dimensions, 2D FFT
>>> x = torch.randn(3, 3, 5, 5, 2)
>>> y = torch.fft(x, 2)
>>> y.shape
torch.Size([3, 3, 5, 5, 2])
```

torch.**ifft**(*input*, *signal_ndim*, *normalized=False*) → Tensor

Complex-to-complex Inverse Discrete Fourier Transform

This method computes the complex-to-complex inverse discrete Fourier transform. Ignoring the batch dimensions, it computes the following expression:

$$X[\omega_1, \dots, \omega_d] = \frac{1}{\prod_{i=1}^{d} N_i} \sum_{n_1=0}^{N_1-1} \cdots \sum_{n_d=0}^{N_d-1} x[n_1, \dots, n_d] e^{\, j \, 2\pi \sum_{i=0}^{d} \frac{\omega_i n_i}{N_i}},$$

where d = `signal_ndim` is number of dimensions for the signal, and N_i is the size of signal dimension i.

The argument specifications are almost identical with *fft()*. However, if `normalized` is set to `True`, this instead returns the results multiplied by $\sqrt{\prod_{i=1}^{d} N_i}$, to become a unitary operator. Therefore, to invert a *fft()*, the `normalized` argument should be set identically for *fft()*.

Returns the real and the imaginary parts together as one tensor of the same shape of `input`.

The inverse of this function is *fft()*.

Note: For CUDA tensors, an LRU cache is used for cuFFT plans to speed up repeatedly running FFT methods on tensors of same geometry with same same configuration.

Changing `torch.backends.cuda.cufft_plan_cache.max_size` (default is 4096 on CUDA 10 and newer, and 1023 on older CUDA versions) controls the capacity of this cache. Some cuFFT plans may allocate GPU memory. You can use `torch.backends.cuda.cufft_plan_cache.size` to query the number of plans currently in cache, and `torch.backends.cuda.cufft_plan_cache.clear()` to clear the cache.

Warning: For CPU tensors, this method is currently only available with MKL. Use `torch.backends.mkl.is_available()` to check if MKL is installed.

Parameters

- **input** (*Tensor*) – the input tensor of at least `signal_ndim + 1` dimensions
- **signal_ndim** (*int*) – the number of dimensions in each signal. `signal_ndim` can only be 1, 2 or 3
- **normalized** (*bool, optional*) – controls whether to return normalized results. Default: `False`

Returns A tensor containing the complex-to-complex inverse Fourier transform result

Return type *Tensor*

Example:

```
>>> x = torch.randn(3, 3, 2)
>>> x
tensor([[[ 1.2766,  1.3680],
         [-0.8337,  2.0251],
         [ 0.9465, -1.4390]],

        [[-0.1890,  1.6010],
         [ 1.1034, -1.9230],
         [-0.9482,  1.0775]],

        [[-0.7708, -0.8176],
         [-0.1843, -0.2287],
         [-1.9034, -0.2196]]])
>>> y = torch.fft(x, 2)
>>> torch.ifft(y, 2)  # recover x
tensor([[[ 1.2766,  1.3680],
         [-0.8337,  2.0251],
         [ 0.9465, -1.4390]],

        [[-0.1890,  1.6010],
         [ 1.1034, -1.9230],
         [-0.9482,  1.0775]],

        [[-0.7708, -0.8176],
         [-0.1843, -0.2287],
         [-1.9034, -0.2196]]])
```

torch.**rfft**(*input*, *signal_ndim*, *normalized=False*, *onesided=True*) → Tensor

Real-to-complex Discrete Fourier Transform

This method computes the real-to-complex discrete Fourier transform. It is mathematically equivalent with *fft()* with differences only in formats of the input and output.

This method supports 1D, 2D and 3D real-to-complex transforms, indicated by `signal_ndim`. `input` must be a tensor with at least `signal_ndim` dimensions with optionally arbitrary number of leading batch dimensions. If `normalized` is set to `True`, this normalizes the result by dividing it with $\sqrt{\prod_{i=1}^{K} N_i}$ so that the operator is unitary, where N_i is the size of signal dimension i.

The real-to-complex Fourier transform results follow conjugate symmetry:

$$X[\omega_1, \ldots, \omega_d] = X^*[N_1 - \omega_1, \ldots, N_d - \omega_d],$$

where the index arithmetic is computed modulus the size of the corresponding dimension, * is the conjugate operator, and d = `signal_ndim`. `onesided` flag controls whether to avoid redundancy in the output results. If set to `True` (default), the output will not be full complex result of shape $(*, 2)$, where $*$ is the shape of `input`, but instead the last dimension will be halfed as of size $\lfloor \frac{N_d}{2} \rfloor + 1$.

The inverse of this function is *irfft()*.

Note: For CUDA tensors, an LRU cache is used for cuFFT plans to speed up repeatedly running FFT methods on tensors of same geometry with same same configuration.

Changing `torch.backends.cuda.cufft_plan_cache.max_size` (default is 4096 on CUDA 10 and newer, and 1023 on older CUDA versions) controls the capacity of this cache. Some cuFFT plans may allocate GPU memory. You can use `torch.backends.cuda.cufft_plan_cache.size` to query the number of plans currently in cache, and `torch.backends.cuda.cufft_plan_cache.clear()` to clear the cache.

Warning: For CPU tensors, this method is currently only available with MKL. Use `torch.backends.mkl.is_available()` to check if MKL is installed.

Parameters

- **input** (*Tensor*) – the input tensor of at least `signal_ndim` dimensions
- **signal_ndim** (*int*) – the number of dimensions in each signal. `signal_ndim` can only be 1, 2 or 3
- **normalized** (*bool, optional*) – controls whether to return normalized results. Default: `False`
- **onesided** (*bool, optional*) – controls whether to return half of results to avoid redundancy. Default: `True`

Returns A tensor containing the real-to-complex Fourier transform result

Return type *Tensor*

Example:

```
>>> x = torch.randn(5, 5)
>>> torch.rfft(x, 2).shape
torch.Size([5, 3, 2])
>>> torch.rfft(x, 2, onesided=False).shape
torch.Size([5, 5, 2])
```

`torch.irfft`(*input*, *signal_ndim*, *normalized=False*, *onesided=True*, *signal_sizes=None*) → Tensor

Complex-to-real Inverse Discrete Fourier Transform

This method computes the complex-to-real inverse discrete Fourier transform. It is mathematically equivalent with *ifft()* with differences only in formats of the input and output.

The argument specifications are almost identical with *ifft()*. Similar to *ifft()*, if `normalized` is set to `True`, this normalizes the result by multiplying it with $\sqrt{\prod_{i=1}^{K} N_i}$ so that the operator is unitary, where N_i is the size of signal dimension i.

Due to the conjugate symmetry, `input` do not need to contain the full complex frequency values. Roughly half of the values will be sufficient, as is the case when `input` is given by *rfft()* with `rfft(signal, onesided=True)`. In such case, set the `onesided` argument of this method to `True`. Moreover, the original signal shape information can sometimes be lost, optionally set `signal_sizes` to be the size of the original signal (without the batch dimensions if in batched mode) to recover it with correct shape.

Therefore, to invert an *rfft()*, the `normalized` and `onesided` arguments should be set identically for *irfft()*, and preferrably a `signal_sizes` is given to avoid size mismatch. See the example below for a case of size mismatch.

See *rfft()* for details on conjugate symmetry.

The inverse of this function is *rfft()*.

Warning: Generally speaking, the input of this function should contain values following conjugate symmetry. Note that even if `onesided` is `True`, often symmetry on some part is still needed. When this requirement is not satisfied, the behavior of *irfft()* is undefined. Since *torch.autograd.gradcheck()* estimates numerical Jacobian with point perturbations, *irfft()* will almost certainly fail the check.

Note: For CUDA tensors, an LRU cache is used for cuFFT plans to speed up repeatedly running FFT methods on tensors of same geometry with same same configuration.

Changing `torch.backends.cuda.cufft_plan_cache.max_size` (default is 4096 on CUDA 10 and newer, and 1023 on older CUDA versions) controls the capacity of this cache. Some cuFFT plans may allocate GPU memory. You can use `torch.backends.cuda.cufft_plan_cache.size` to query the number of plans currently in cache, and `torch.backends.cuda.cufft_plan_cache.clear()` to clear the cache.

Warning: For CPU tensors, this method is currently only available with MKL. Use `torch.backends.mkl.is_available()` to check if MKL is installed.

Parameters

- **input** (*Tensor*) – the input tensor of at least `signal_ndim + 1` dimensions
- **signal_ndim** (*int*) – the number of dimensions in each signal. `signal_ndim` can only be 1, 2 or 3
- **normalized** (*bool, optional*) – controls whether to return normalized results. Default: `False`
- **onesided** (*bool, optional*) – controls whether `input` was halfed to avoid redundancy, e.g., by *rfft()*. Default: `True`
- **signal_sizes** (list or `torch.Size`, optional) – the size of the original signal (without batch dimension). Default: `None`

Returns A tensor containing the complex-to-real inverse Fourier transform result

Return type *Tensor*

Example:

```
>>> x = torch.randn(4, 4)
>>> torch.rfft(x, 2, onesided=True).shape
torch.Size([4, 3, 2])
>>>
>>> # notice that with onesided=True, output size does not determine the original␣
↪signal size
>>> x = torch.randn(4, 5)

>>> torch.rfft(x, 2, onesided=True).shape
torch.Size([4, 3, 2])
>>>
>>> # now we use the original shape to recover x
>>> x
tensor([[-0.8992,  0.6117, -1.6091, -0.4155, -0.8346],
        [-2.1596, -0.0853,  0.7232,  0.1941, -0.0789],
        [-2.0329,  1.1031,  0.6869, -0.5042,  0.9895],
        [-0.1884,  0.2858, -1.5831,  0.9917, -0.8356]])
>>> y = torch.rfft(x, 2, onesided=True)
>>> torch.irfft(y, 2, onesided=True, signal_sizes=x.shape)  # recover x
tensor([[-0.8992,  0.6117, -1.6091, -0.4155, -0.8346],
        [-2.1596, -0.0853,  0.7232,  0.1941, -0.0789],
```

(continues on next page)

(continued from previous page)

```
        [-2.0329,  1.1031,  0.6869, -0.5042,  0.9895],
        [-0.1884,  0.2858, -1.5831,  0.9917, -0.8356]])
```

torch.**stft**(*input*, *n_fft*, *hop_length=None*, *win_length=None*, *window=None*, *center=True*, *pad_mode='reflect'*, *normalized=False*, *onesided=True*)

Short-time Fourier transform (STFT).

Ignoring the optional batch dimension, this method computes the following expression:

$$X[m, \omega] = \sum_{k=0}^{\text{win\_length-1}} \text{window}[k]\ \text{input}[m \times \text{hop\_length} + k]\ \exp\left(-j \frac{2\pi \cdot \omega k}{\text{win\_length}}\right),$$

where m is the index of the sliding window, and ω is the frequency that $0 \leq \omega < \text{n\_fft}$. When `onesided` is the default value `True`,

- `input` must be either a 1-D time sequence or a 2-D batch of time sequences.
- If `hop_length` is `None` (default), it is treated as equal to `floor(n_fft / 4)`.
- If `win_length` is `None` (default), it is treated as equal to `n_fft`.
- `window` can be a 1-D tensor of size `win_length`, e.g., from *`torch.hann_window()`*. If `window` is `None` (default), it is treated as if having 1 everywhere in the window. If win_length $<$ n_fft, `window` will be padded on both sides to length `n_fft` before being applied.
- If `center` is `True` (default), `input` will be padded on both sides so that the t-th frame is centered at time $t \times$ hop_length. Otherwise, the t-th frame begins at time $t \times$ hop_length.
- `pad_mode` determines the padding method used on `input` when `center` is `True`. See *`torch.nn.functional.pad()`* for all available options. Default is `"reflect"`.
- If `onesided` is `True` (default), only values for ω in $\left[0, 1, 2, \ldots, \left\lfloor \frac{\text{n\_fft}}{2} \right\rfloor + 1\right]$ are returned because the real-to-complex Fourier transform satisfies the conjugate symmetry, i.e., $X[m, \omega] = X[m, \text{n\_fft} - \omega]^*$.
- If `normalized` is `True` (default is `False`), the function returns the normalized STFT results, i.e., multiplied by $(\text{frame\_length})^{-0.5}$.

Returns the real and the imaginary parts together as one tensor of size $(* \times N \times T \times 2)$, where $*$ is the optional batch size of `input`, N is the number of frequencies where STFT is applied, T is the total number of frames used, and each pair in the last dimension represents a complex number as the real part and the imaginary part.

Warning: This function changed signature at version 0.4.1. Calling with the previous signature may cause error or return incorrect result.

Parameters

- **input** (*Tensor*) – the input tensor
- **n_fft** (*int*) – size of Fourier transform
- **hop_length** (*int, optional*) – the distance between neighboring sliding window frames. Default: `None` (treated as equal to `floor(n_fft / 4)`)
- **win_length** (*int, optional*) – the size of window frame and STFT filter. Default: `None` (treated as equal to `n_fft`)
- **window** (*Tensor, optional*) – the optional window function. Default: `None` (treated as window of all 1 s)

- **center** (*bool, optional*) – whether to pad `input` on both sides so that the t-th frame is centered at time $t \times$ hop_length. Default: `True`
- **pad_mode** (*string, optional*) – controls the padding method used when `center` is `True`. Default: `"reflect"`
- **normalized** (*bool, optional*) – controls whether to return the normalized STFT results Default: `False`
- **onesided** (*bool, optional*) – controls whether to return half of results to avoid redundancy Default: `True`

Returns A tensor containing the STFT result with shape described above

Return type *Tensor*

`torch.`**bartlett_window**(*window_length*, *periodic=True*, *dtype=None*, *layout=torch.strided*, *device=None*, *requires_grad=False*) → Tensor

Bartlett window function.

$$w[n] = 1 - \left| \frac{2n}{N-1} - 1 \right| = \begin{cases} \frac{2n}{N-1} & \text{if } 0 \leq n \leq \frac{N-1}{2} \\ 2 - \frac{2n}{N-1} & \text{if } \frac{N-1}{2} < n < N \end{cases},$$

where N is the full window size.

The input `window_length` is a positive integer controlling the returned window size. `periodic` flag determines whether the returned window trims off the last duplicate value from the symmetric window and is ready to be used as a periodic window with functions like *`torch.stft()`*. Therefore, if `periodic` is true, the N in above formula is in fact window_length + 1. Also, we always have `torch.bartlett_window(L, periodic=True)` equal to `torch.bartlett_window(L + 1, periodic=False)[:-1])`.

Note: If `window_length` $= 1$, the returned window contains a single value 1.

Parameters

- **window_length** (*int*) – the size of returned window
- **periodic** (*bool, optional*) – If True, returns a window to be used as periodic function. If False, return a symmetric window.
- **dtype** (*torch.dtype*, optional) – the desired data type of returned tensor. Default: if `None`, uses a global default (see *`torch.set_default_tensor_type()`*). Only floating point types are supported.
- **layout** (*torch.layout*, optional) – the desired layout of returned window tensor. Only `torch.strided` (dense layout) is supported.
- **device** (*torch.device*, optional) – the desired device of returned tensor. Default: if `None`, uses the current device for the default tensor type (see *`torch.set_default_tensor_type()`*). `device` will be the CPU for CPU tensor types and the current CUDA device for CUDA tensor types.
- **requires_grad** (*bool, optional*) – If autograd should record operations on the returned tensor. Default: `False`.

Returns A 1-D tensor of size (window_length,) containing the window

Return type *Tensor*

torch.**blackman_window**(*window_length*, *periodic=True*, *dtype=None*, *layout=torch.strided*, *device=None*, *requires_grad=False*) → Tensor

Blackman window function.

$$w[n] = 0.42 - 0.5 \cos\left(\frac{2\pi n}{N-1}\right) + 0.08 \cos\left(\frac{4\pi n}{N-1}\right)$$

where N is the full window size.

The input `window_length` is a positive integer controlling the returned window size. `periodic` flag determines whether the returned window trims off the last duplicate value from the symmetric window and is ready to be used as a periodic window with functions like *`torch.stft()`*. Therefore, if `periodic` is true, the N in above formula is in fact window_length + 1. Also, we always have `torch.blackman_window(L, periodic=True)` equal to `torch.blackman_window(L + 1, periodic=False)[:-1])`.

Note: If `window_length` $= 1$, the returned window contains a single value 1.

Parameters

- **window_length** (*int*) – the size of returned window
- **periodic** (*bool, optional*) – If True, returns a window to be used as periodic function. If False, return a symmetric window.
- **dtype** (*torch.dtype*, optional) – the desired data type of returned tensor. Default: if `None`, uses a global default (see *`torch.set_default_tensor_type()`*). Only floating point types are supported.
- **layout** (*torch.layout*, optional) – the desired layout of returned window tensor. Only `torch.strided` (dense layout) is supported.
- **device** (*torch.device*, optional) – the desired device of returned tensor. Default: if `None`, uses the current device for the default tensor type (see *`torch.set_default_tensor_type()`*). `device` will be the CPU for CPU tensor types and the current CUDA device for CUDA tensor types.
- **requires_grad** (*bool, optional*) – If autograd should record operations on the returned tensor. Default: `False`.

Returns A 1-D tensor of size (window_length,) containing the window

Return type *Tensor*

torch.**hamming_window**(*window_length*, *periodic=True*, *alpha=0.54*, *beta=0.46*, *dtype=None*, *layout=torch.strided*, *device=None*, *requires_grad=False*) → Tensor

Hamming window function.

$$w[n] = \alpha - \beta\ \cos\left(\frac{2\pi n}{N-1}\right),$$

where N is the full window size.

The input `window_length` is a positive integer controlling the returned window size. `periodic` flag determines whether the returned window trims off the last duplicate value from the symmetric window and is ready to be used as a periodic window with functions like *`torch.stft()`*. Therefore, if `periodic` is true, the N in above formula is in fact window_length + 1. Also, we always have `torch.hamming_window(L, periodic=True)` equal to `torch.hamming_window(L + 1, periodic=False)[:-1])`.

Note: If `window_length` $= 1$, the returned window contains a single value 1.

Note: This is a generalized version of *torch.hann_window()*.

Parameters

- **window_length** (*int*) – the size of returned window
- **periodic** (*bool, optional*) – If True, returns a window to be used as periodic function. If False, return a symmetric window.
- **dtype** (*torch.dtype*, optional) – the desired data type of returned tensor. Default: if `None`, uses a global default (see *torch.set_default_tensor_type()*). Only floating point types are supported.
- **layout** (*torch.layout*, optional) – the desired layout of returned window tensor. Only `torch.strided` (dense layout) is supported.
- **device** (*torch.device*, optional) – the desired device of returned tensor. Default: if `None`, uses the current device for the default tensor type (see *torch.set_default_tensor_type()*). `device` will be the CPU for CPU tensor types and the current CUDA device for CUDA tensor types.
- **requires_grad** (*bool, optional*) – If autograd should record operations on the returned tensor. Default: `False`.

Returns A 1-D tensor of size (window_length,) containing the window

Return type *Tensor*

`torch.`**hann_window**(*window_length*, *periodic=True*, *dtype=None*, *layout=torch.strided*, *device=None*, *requires_grad=False*) → Tensor

Hann window function.

$$w[n] = \frac{1}{2}\left[1 - \cos\left(\frac{2\pi n}{N-1}\right)\right] = \sin^2\left(\frac{\pi n}{N-1}\right),$$

where N is the full window size.

The input `window_length` is a positive integer controlling the returned window size. `periodic` flag determines whether the returned window trims off the last duplicate value from the symmetric window and is ready to be used as a periodic window with functions like *torch.stft()*. Therefore, if `periodic` is true, the N in above formula is in fact window_length $+ 1$. Also, we always have `torch.hann_window(L, periodic=True)` equal to `torch.hann_window(L + 1, periodic=False)[:-1])`.

Note: If `window_length` $= 1$, the returned window contains a single value 1.

Parameters

- **window_length** (*int*) – the size of returned window
- **periodic** (*bool, optional*) – If True, returns a window to be used as periodic function. If False, return a symmetric window.

- **dtype** (*torch.dtype*, optional) – the desired data type of returned tensor. Default: if None, uses a global default (see *torch.set_default_tensor_type()*). Only floating point types are supported.
- **layout** (*torch.layout*, optional) – the desired layout of returned window tensor. Only torch.strided (dense layout) is supported.
- **device** (*torch.device*, optional) – the desired device of returned tensor. Default: if None, uses the current device for the default tensor type (see *torch.set_default_tensor_type()*). device will be the CPU for CPU tensor types and the current CUDA device for CUDA tensor types.
- **requires_grad** (*bool, optional*) – If autograd should record operations on the returned tensor. Default: False.

Returns A 1-D tensor of size (window_length,) containing the window

Return type *Tensor*

15.7.5 Other Operations

torch.**bincount**(*self*, *weights=None*, *minlength=0*) → Tensor

Count the frequency of each value in an array of non-negative ints.

The number of bins (size 1) is one larger than the largest value in input unless input is empty, in which case the result is a tensor of size 0. If minlength is specified, the number of bins is at least minlength and if input is empty, then the result is tensor of size minlength filled with zeros. If n is the value at position i, out[n] += weights[i] if weights is specified else out[n] += 1.

Note: When using the CUDA backend, this operation may induce nondeterministic behaviour that is not easily switched off. Please see the notes on *Reproducibility* for background.

Parameters

- **input** (Tensor) – 1-d int tensor
- **weights** (Tensor) – optional, weight for each value in the input tensor. Should be of same size as input tensor.
- **minlength** (*int*) – optional, minimum number of bins. Should be non-negative.

Returns a tensor of shape Size([max(input) + 1]) if input is non-empty, else Size(0)

Return type output (*Tensor*)

Example:

```
>>> input = torch.randint(0, 8, (5,), dtype=torch.int64)
>>> weights = torch.linspace(0, 1, steps=5)
>>> input, weights
(tensor([4, 3, 6, 3, 4]),
 tensor([ 0.0000,  0.2500,  0.5000,  0.7500,  1.0000])

>>> torch.bincount(input)
tensor([0, 0, 0, 2, 2, 0, 1])

>>> input.bincount(weights)
tensor([0.0000, 0.0000, 0.0000, 1.0000, 1.0000, 0.0000, 0.5000])
```

torch.**broadcast_tensors**(**tensors*) → List of Tensors

Broadcasts the given tensors according to *Broadcasting semantics*.

Parameters ***tensors** – any number of tensors of the same type

> **Warning:** More than one element of a broadcasted tensor may refer to a single memory location. As a result, in-place operations (especially ones that are vectorized) may result in incorrect behavior. If you need to write to the tensors, please clone them first.

Example:

```
>>> x = torch.arange(3).view(1, 3)
>>> y = torch.arange(2).view(2, 1)
>>> a, b = torch.broadcast_tensors(x, y)
>>> a.size()
torch.Size([2, 3])
>>> a
tensor([[0, 1, 2],
        [0, 1, 2]])
```

torch.**cartesian_prod**(**tensors*)

Do cartesian product of the given sequence of tensors. The behavior is similar to pythons *itertools.product*.

Parameters ***tensors** – any number of 1 dimensional tensors.

Returns

A tensor equivalent to converting all the input tensors into lists, do *itertools.product* on these lists, and finally convert the resulting list into tensor.

Return type *Tensor*

Example:

```
>>> a = [1, 2, 3]
>>> b = [4, 5]
>>> list(itertools.product(a, b))
[(1, 4), (1, 5), (2, 4), (2, 5), (3, 4), (3, 5)]
>>> tensor_a = torch.tensor(a)
>>> tensor_b = torch.tensor(b)
>>> torch.cartesian_prod(tensor_a, tensor_b)
tensor([[1, 4],
        [1, 5],
        [2, 4],
        [2, 5],
        [3, 4],
        [3, 5]])
```

torch.**cdist**()

torch.**combinations**(*tensor*, *r=2*, *with_replacement=False*) → seq

Compute combinations of length *r* of the given tensor. The behavior is similar to pythons *itertools.combinations* when *with_replacement* is set to *False*, and *itertools.combinations_with_replacement* when *with_replacement* is set to *True*.

Parameters

- **tensor** (Tensor) – 1D vector.
- **r** (*int, optional*) – number of elements to combine

- **with_replacement** (*boolean, optional*) – whether to allow duplication in combination

Returns A tensor equivalent to converting all the input tensors into lists, do *itertools.combinations* or *itertools.combinations_with_replacement* on these lists, and finally convert the resulting list into tensor.

Return type *Tensor*

Example:

```
>>> a = [1, 2, 3]
>>> list(itertools.combinations(a, r=2))
[(1, 2), (1, 3), (2, 3)]
>>> list(itertools.combinations(a, r=3))
[(1, 2, 3)]
>>> list(itertools.combinations_with_replacement(a, r=2))
[(1, 1), (1, 2), (1, 3), (2, 2), (2, 3), (3, 3)]
>>> tensor_a = torch.tensor(a)
>>> torch.combinations(tensor_a)
tensor([[1, 2],
        [1, 3],
        [2, 3]])
>>> torch.combinations(tensor_a, r=3)
tensor([[1, 2, 3]])
>>> torch.combinations(tensor_a, with_replacement=True)
tensor([[1, 1],
        [1, 2],
        [1, 3],
        [2, 2],
        [2, 3],
        [3, 3]])
```

`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.

Parameters

- **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 output tensor

Example:

```
>>> a = torch.randn(4, 3)
>>> a
tensor([[-0.3956,  1.1455,  1.6895],
        [-0.5849,  1.3672,  0.3599],
        [-1.1626,  0.7180, -0.0521],
        [-0.1339,  0.9902, -2.0225]])
>>> b = torch.randn(4, 3)
>>> b
tensor([[-0.0257, -1.4725, -1.2251],
        [-1.1479, -0.7005, -1.9757],
```

(continues on next page)

(continued from previous page)

```
        [-1.3904,  0.3726, -1.1836],
        [-0.9688, -0.7153,  0.2159]])
>>> torch.cross(a, b, dim=1)
tensor([[ 1.0844, -0.5281,  0.6120],
        [-2.4490, -1.5687,  1.9792],
        [-0.8304, -1.3037,  0.5650],
        [-1.2329,  1.9883,  1.0551]])
>>> torch.cross(a, b)
tensor([[ 1.0844, -0.5281,  0.6120],
        [-2.4490, -1.5687,  1.9792],
        [-0.8304, -1.3037,  0.5650],
        [-1.2329,  1.9883,  1.0551]])
```

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

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

The argument *diagonal* controls which diagonal to consider:

- If *diagonal* = 0, it is the main diagonal.
- If *diagonal* > 0, it is above the main diagonal.
- If *diagonal* < 0, it is below the main diagonal.

Parameters

- **input** (Tensor) – the input tensor
- **diagonal** (*int, optional*) – the diagonal to consider
- **out** (Tensor, *optional*) – the output tensor

See also:

torch.diagonal() always returns the diagonal of its input.

torch.diagflat() always constructs a tensor with diagonal elements specified by the input.

Examples:

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

```
>>> a = torch.randn(3)
>>> a
tensor([ 0.5950,-0.0872, 2.3298])
>>> torch.diag(a)
tensor([[ 0.5950, 0.0000, 0.0000],
        [ 0.0000,-0.0872, 0.0000],
        [ 0.0000, 0.0000, 2.3298]])
>>> torch.diag(a, 1)
tensor([[ 0.0000, 0.5950, 0.0000, 0.0000],
        [ 0.0000, 0.0000,-0.0872, 0.0000],
        [ 0.0000, 0.0000, 0.0000, 2.3298],
        [ 0.0000, 0.0000, 0.0000, 0.0000]])
```

Get the k-th diagonal of a given matrix:

```
>>> a = torch.randn(3, 3)
>>> a
tensor([[-0.4264, 0.0255,-0.1064],
        [ 0.8795,-0.2429, 0.1374],
        [ 0.1029,-0.6482,-1.6300]])
>>> torch.diag(a, 0)
tensor([-0.4264,-0.2429,-1.6300])
>>> torch.diag(a, 1)
tensor([ 0.0255, 0.1374])
```

torch.**diag_embed**(*input*, *offset=0*, *dim1=-2*, *dim2=-1*) → Tensor

Creates a tensor whose diagonals of certain 2D planes (specified by `dim1` and `dim2`) are filled by `input`. To facilitate creating batched diagonal matrices, the 2D planes formed by the last two dimensions of the returned tensor are chosen by default.

The argument `offset` controls which diagonal to consider:

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

The size of the new matrix will be calculated to make the specified diagonal of the size of the last input dimension. Note that for `offset` other than 0, the order of `dim1` and `dim2` matters. Exchanging them is equivalent to changing the sign of `offset`.

Applying *torch.diagonal()* to the output of this function with the same arguments yields a matrix identical to input. However, *torch.diagonal()* has different default dimensions, so those need to be explicitly specified.

Parameters

- **input** (Tensor) – the input tensor. Must be at least 1-dimensional.
- **offset** (*int, optional*) – which diagonal to consider. Default: 0 (main diagonal).
- **dim1** (*int, optional*) – first dimension with respect to which to take diagonal. Default: -2.
- **dim2** (*int, optional*) – second dimension with respect to which to take diagonal. Default: -1.

Example:

```
>>> a = torch.randn(2, 3)
>>> torch.diag_embed(a)
tensor([[[ 1.5410,  0.0000,  0.0000],
         [ 0.0000, -0.2934,  0.0000],
         [ 0.0000,  0.0000, -2.1788]],

        [[ 0.5684,  0.0000,  0.0000],
         [ 0.0000, -1.0845,  0.0000],
         [ 0.0000,  0.0000, -1.3986]]])

>>> torch.diag_embed(a, offset=1, dim1=0, dim2=2)
tensor([[[ 0.0000,  1.5410,  0.0000,  0.0000],
         [ 0.0000,  0.5684,  0.0000,  0.0000]],

        [[ 0.0000,  0.0000, -0.2934,  0.0000],
         [ 0.0000,  0.0000, -1.0845,  0.0000]],
```

(continues on next page)

(continued from previous page)

```
        [[ 0.0000,  0.0000,  0.0000, -2.1788],
         [ 0.0000,  0.0000,  0.0000, -1.3986]],

        [[ 0.0000,  0.0000,  0.0000,  0.0000],
         [ 0.0000,  0.0000,  0.0000,  0.0000]]])
```

torch.**diagflat**(*input*, *diagonal=0*) → Tensor

- If `input` is a vector (1-D tensor), then returns a 2-D square tensor with the elements of `input` as the diagonal.
- If `input` is a tensor with more than one dimension, then returns a 2-D tensor with diagonal elements equal to a flattened `input`.

The argument `offset` controls which diagonal to consider:

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

Parameters

- **input** (Tensor) – the input tensor
- **offset** (*int, optional*) – the diagonal to consider. Default: 0 (main diagonal).

Examples:

```
>>> a = torch.randn(3)
>>> a
tensor([-0.2956, -0.9068,  0.1695])
>>> torch.diagflat(a)
tensor([[-0.2956,  0.0000,  0.0000],
        [ 0.0000, -0.9068,  0.0000],
        [ 0.0000,  0.0000,  0.1695]])
>>> torch.diagflat(a, 1)
tensor([[ 0.0000, -0.2956,  0.0000,  0.0000],
        [ 0.0000,  0.0000, -0.9068,  0.0000],
        [ 0.0000,  0.0000,  0.0000,  0.1695],
        [ 0.0000,  0.0000,  0.0000,  0.0000]])

>>> a = torch.randn(2, 2)
>>> a
tensor([[ 0.2094, -0.3018],
        [-0.1516,  1.9342]])
>>> torch.diagflat(a)
tensor([[ 0.2094,  0.0000,  0.0000,  0.0000],
        [ 0.0000, -0.3018,  0.0000,  0.0000],
        [ 0.0000,  0.0000, -0.1516,  0.0000],
        [ 0.0000,  0.0000,  0.0000,  1.9342]])
```

torch.**diagonal**(*input*, *offset=0*, *dim1=0*, *dim2=1*) → Tensor

Returns a partial view of `input` with the its diagonal elements with respect to `dim1` and `dim2` appended as a dimension at the end of the shape.

The argument `offset` controls which diagonal to consider:

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

Applying *torch.diag_embed()* to the output of this function with the same arguments yields a diagonal matrix with the diagonal entries of the input. However, *torch.diag_embed()* has different default dimensions, so those need to be explicitly specified.

Parameters

- **input** (*Tensor*) – the input tensor. Must be at least 2-dimensional.
- **offset** (*int, optional*) – which diagonal to consider. Default: 0 (main diagonal).
- **dim1** (*int, optional*) – first dimension with respect to which to take diagonal. Default: 0.
- **dim2** (*int, optional*) – second dimension with respect to which to take diagonal. Default: 1.

Note: To take a batch diagonal, pass in dim1=-2, dim2=-1.

Examples:

```
>>> a = torch.randn(3, 3)
>>> a
tensor([[-1.0854,  1.1431, -0.1752],
        [ 0.8536, -0.0905,  0.0360],
        [ 0.6927, -0.3735, -0.4945]])


>>> torch.diagonal(a, 0)
tensor([-1.0854, -0.0905, -0.4945])


>>> torch.diagonal(a, 1)
tensor([ 1.1431,  0.0360])


>>> x = torch.randn(2, 5, 4, 2)
>>> torch.diagonal(x, offset=-1, dim1=1, dim2=2)
tensor([[[-1.2631,  0.3755, -1.5977, -1.8172],
         [-1.1065,  1.0401, -0.2235, -0.7938]],

        [[-1.7325, -0.3081,  0.6166,  0.2335],
         [ 1.0500,  0.7336, -0.3836, -1.1015]]])
```

`torch.`**`einsum`**(*equation*, **operands*) → Tensor

This function provides a way of computing multilinear expressions (i.e. sums of products) using the Einstein summation convention.

Parameters

- **equation** (*string*) – The equation is given in terms of lower case letters (indices) to be associated with each dimension of the operands and result. The left hand side lists the operands dimensions, separated by commas. There should be one index letter per tensor dimension. The right hand side follows after -> and gives the indices for the output. If the -> and right hand side are omitted, it implicitly defined as the alphabetically sorted list of all

indices appearing exactly once in the left hand side. The indices not apprearing in the output are summed over after multiplying the operands entries. If an index appears several times for the same operand, a diagonal is taken. Ellipses represent a fixed number of dimensions. If the right hand side is inferred, the ellipsis dimensions are at the beginning of the output.

- **operands** (*list of Tensors*) – The operands to compute the Einstein sum of.

Examples:

```
>>> x = torch.randn(5)
>>> y = torch.randn(4)
>>> torch.einsum('i,j->ij', x, y)  # outer product
tensor([[-0.0570, -0.0286, -0.0231,  0.0197],
        [ 1.2616,  0.6335,  0.5113, -0.4351],
        [ 1.4452,  0.7257,  0.5857, -0.4984],
        [-0.4647, -0.2333, -0.1883,  0.1603],
        [-1.1130, -0.5588, -0.4510,  0.3838]])


>>> A = torch.randn(3,5,4)
>>> l = torch.randn(2,5)
>>> r = torch.randn(2,4)
>>> torch.einsum('bn,anm,bm->ba', l, A, r) # compare torch.nn.functional.bilinear
tensor([[-0.3430, -5.2405,  0.4494],
        [ 0.3311,  5.5201, -3.0356]])


>>> As = torch.randn(3,2,5)
>>> Bs = torch.randn(3,5,4)
>>> torch.einsum('bij,bjk->bik', As, Bs) # batch matrix multiplication
tensor([[[-1.0564, -1.5904,  3.2023,  3.1271],
         [-1.6706, -0.8097, -0.8025, -2.1183]],

        [[ 4.2239,  0.3107, -0.5756, -0.2354],
         [-1.4558, -0.3460,  1.5087, -0.8530]],

        [[ 2.8153,  1.8787, -4.3839, -1.2112],
         [ 0.3728, -2.1131,  0.0921,  0.8305]]])

>>> A = torch.randn(3, 3)
>>> torch.einsum('ii->i', A) # diagonal
tensor([-0.7825,  0.8291, -0.1936])

>>> A = torch.randn(4, 3, 3)
>>> torch.einsum('...ii->...i', A) # batch diagonal
tensor([[-1.0864,  0.7292,  0.0569],
        [-0.9725, -1.0270,  0.6493],
        [ 0.5832, -1.1716, -1.5084],
        [ 0.4041, -1.1690,  0.8570]])

>>> A = torch.randn(2, 3, 4, 5)
>>> torch.einsum('...ij->...ji', A).shape # batch permute
torch.Size([2, 3, 5, 4])
```

torch.**flatten**(*input*, *start_dim=0*, *end_dim=-1*) → Tensor

Flattens a contiguous range of dims in a tensor.

Parameters

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

- **start_dim** (*int*) – the first dim to flatten
- **end_dim** (*int*) – the last dim to flatten

Example:

```
>>> t = torch.tensor([[[1, 2],
                       [3, 4]],
                      [[5, 6],
                       [7, 8]]])
>>> torch.flatten(t)
tensor([1, 2, 3, 4, 5, 6, 7, 8])
>>> torch.flatten(t, start_dim=1)
tensor([[1, 2, 3, 4],
        [5, 6, 7, 8]])
```

torch.**flip**(*input*, *dims*) → Tensor

Reverse the order of a n-D tensor along given axis in dims.

Parameters

- **input** (Tensor) – the input tensor
- **dims** (*a list or tuple*) – axis to flip on

Example:

```
>>> x = torch.arange(8).view(2, 2, 2)
>>> x
tensor([[[ 0,  1],
         [ 2,  3]],

        [[ 4,  5],
         [ 6,  7]]])
>>> torch.flip(x, [0, 1])
tensor([[[ 6,  7],
         [ 4,  5]],

        [[ 2,  3],
         [ 0,  1]]])
```

torch.**rot90**(*input*, *k*, *dims*) → Tensor

Rotate a n-D tensor by 90 degrees in the plane specified by dims axis. Rotation direction is from the first towards the second axis if k > 0, and from the second towards the first for k < 0.

Parameters

- **input** (Tensor) – the input tensor
- **k** (*int*) – number of times to rotate
- **dims** (*a list or tuple*) – axis to rotate

Example:

```
>>> x = torch.arange(4).view(2, 2)
>>> x
tensor([[0, 1],
        [2, 3]])
>>> torch.rot90(x, 1, [0, 1])
tensor([[1, 3],
        [0, 2]])
```

(continues on next page)

(continued from previous page)

```
>>> x = torch.arange(8).view(2, 2, 2)
>>> x
tensor([[[0, 1],
         [2, 3]],

        [[4, 5],
         [6, 7]]])
>>> torch.rot90(x, 1, [1, 2])
tensor([[[1, 3],
         [0, 2]],

        [[5, 7],
         [4, 6]]])
```

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.

Parameters

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

Returns Histogram represented as a tensor

Return type *Tensor*

Example:

```
>>> torch.histc(torch.tensor([1., 2, 1]), bins=4, min=0, max=3)
tensor([ 0.,  2.,  1.,  0.])
```

torch.**meshgrid**(**tensors*, ***kwargs*)

Take N tensors, each of which can be either scalar or 1-dimensional vector, and create N N-dimensional grids, where the i th grid is defined by expanding the i th input over dimensions defined by other inputs.

Args: tensors (list of Tensor): list of scalars or 1 dimensional tensors. Scalars will be treated as tensors of size $(1,)$ automatically

Returns: seq (sequence of Tensors): If the input has k tensors of size $(N_1,), (N_2,), \ldots, (N_k,)$, then the output would also has k tensors, where all tensors are of size $(N_1, N_2, \ldots, N_k)$.

Example:

```
>>> x = torch.tensor([1, 2, 3])
>>> y = torch.tensor([4, 5, 6])
>>> grid_x, grid_y = torch.meshgrid(x, y)
>>> grid_x
tensor([[1, 1, 1],
        [2, 2, 2],
        [3, 3, 3]])
```

(continues on next page)

(continued from previous page)

```
>>> grid_y
tensor([[4, 5, 6],
        [4, 5, 6],
        [4, 5, 6]])
```

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`

Note: If the norm of a row is lower than *maxnorm*, the row is unchanged

Parameters

- **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*) – the output tensor

Example:

```
>>> x = torch.ones(3, 3)
>>> x[1].fill_(2)
tensor([ 2.,  2.,  2.])
>>> x[2].fill_(3)
tensor([ 3.,  3.,  3.])
>>> x
tensor([[ 1.,  1.,  1.],
        [ 2.,  2.,  2.],
        [ 3.,  3.,  3.]])
>>> torch.renorm(x, 1, 0, 5)
tensor([[ 1.0000,  1.0000,  1.0000],
        [ 1.6667,  1.6667,  1.6667],
        [ 1.6667,  1.6667,  1.6667]])
```

torch.**roll**(*input*, *shifts*, *dims=None*) → Tensor

Roll the tensor along the given dimension(s). Elements that are shifted beyond the last position are re-introduced at the first position. If a dimension is not specified, the tensor will be flattened before rolling and then restored to the original shape.

Parameters

- **input** (Tensor) – the input tensor
- **shifts** (*int or tuple of python:ints*) – The number of places by which the elements of the tensor are shifted. If shifts is a tuple, dims must be a tuple of the same size, and each dimension will be rolled by the corresponding value
- **dims** (*int or tuple of python:ints*) – Axis along which to roll

Example:

```
>>> x = torch.tensor([1, 2, 3, 4, 5, 6, 7, 8]).view(4, 2)
>>> x
tensor([[1, 2],
        [3, 4],
        [5, 6],
        [7, 8]])
>>> torch.roll(x, 1, 0)
tensor([[7, 8],
        [1, 2],
        [3, 4],
        [5, 6]])
>>> torch.roll(x, -1, 0)
tensor([[3, 4],
        [5, 6],
        [7, 8],
        [1, 2]])
>>> torch.roll(x, shifts=(2, 1), dims=(0, 1))
tensor([[6, 5],
        [8, 7],
        [2, 1],
        [4, 3]])
```

torch.**tensordot**(*a*, *b*, *dims=2*)

Returns a contraction of a and b over multiple dimensions.

tensordot implements a generalizes the matrix product.

Parameters

- **a** (Tensor) – Left tensor to contract
- **b** (Tensor) – Right tensor to contract
- **dims** (*int or tuple of two lists of python:integers*) – number of dimensions to contract or explicit lists of dimensions for a and b respectively

When called with an integer argument dims = d, and the number of dimensions of a and b is m and n, respectively, it computes

$$r_{i_0,\ldots,i_{m-d},i_d,\ldots,i_n} = \sum_{k_0,\ldots,k_{d-1}} a_{i_0,\ldots,i_{m-d},k_0,\ldots,k_{d-1}} \times b_{k_0,\ldots,k_{d-1},i_d,\ldots,i_n}.$$

When called with dims of the list form, the given dimensions will be contracted in place of the last d of a and the first d of b. The sizes in these dimensions must match, but *tensordot* will deal with broadcasted dimensions.

Examples:

```
>>> a = torch.arange(60.).reshape(3, 4, 5)
>>> b = torch.arange(24.).reshape(4, 3, 2)
>>> torch.tensordot(a, b, dims=([1, 0], [0, 1]))
tensor([[4400., 4730.],
        [4532., 4874.],
        [4664., 5018.],
        [4796., 5162.],
        [4928., 5306.]])

>>> a = torch.randn(3, 4, 5, device='cuda')
>>> b = torch.randn(4, 5, 6, device='cuda')
```

(continues on next page)

(continued from previous page)

```
>>> c = torch.tensordot(a, b, dims=2).cpu()
tensor([[ 8.3504, -2.5436,  6.2922,  2.7556, -1.0732,  3.2741],
        [ 3.3161,  0.0704,  5.0187, -0.4079, -4.3126,  4.8744],
        [ 0.8223,  3.9445,  3.2168, -0.2400,  3.4117,  1.7780]])
```

torch.**trace**(*input*) → Tensor

Returns the sum of the elements of the diagonal of the input 2-D matrix.

Example:

```
>>> x = torch.arange(1., 10.).view(3, 3)
>>> x
tensor([[ 1.,  2.,  3.],
        [ 4.,  5.,  6.],
        [ 7.,  8.,  9.]])
>>> torch.trace(x)
tensor(15.)
```

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

Returns the lower triangular part of the matrix (2-D tensor) or batch of matrices `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 *diagonal* controls which diagonal to consider. If *diagonal* = 0, all elements on and below the main diagonal are retained. A positive value includes just as many diagonals above the main diagonal, and similarly a negative value excludes just as many diagonals below the main diagonal. The main diagonal are the set of indices $\{(i, i)\}$ for $i \in [0, \min\{d_1, d_2\} - 1]$ where d_1, d_2 are the dimensions of the matrix.

Parameters

- **input** (Tensor) – the input tensor
- **diagonal** (*int, optional*) – the diagonal to consider
- **out** (Tensor, *optional*) – the output tensor

Example:

```
>>> a = torch.randn(3, 3)
>>> a
tensor([[-1.0813, -0.8619,  0.7105],
        [ 0.0935,  0.1380,  2.2112],
        [-0.3409, -0.9828,  0.0289]])
>>> torch.tril(a)
tensor([[-1.0813,  0.0000,  0.0000],
        [ 0.0935,  0.1380,  0.0000],
        [-0.3409, -0.9828,  0.0289]])

>>> b = torch.randn(4, 6)
>>> b
tensor([[ 1.2219,  0.5653, -0.2521, -0.2345,  1.2544,  0.3461],
        [ 0.4785, -0.4477,  0.6049,  0.6368,  0.8775,  0.7145],
        [ 1.1502,  3.2716, -1.1243, -0.5413,  0.3615,  0.6864],
        [-0.0614, -0.7344, -1.3164, -0.7648, -1.4024,  0.0978]])
>>> torch.tril(b, diagonal=1)
tensor([[ 1.2219,  0.5653,  0.0000,  0.0000,  0.0000,  0.0000],
        [ 0.4785, -0.4477,  0.6049,  0.0000,  0.0000,  0.0000],
        [ 1.1502,  3.2716, -1.1243, -0.5413,  0.0000,  0.0000],
```

(continues on next page)

(continued from previous page)

```
        [-0.0614, -0.7344, -1.3164, -0.7648, -1.4024,  0.0000]])
>>> torch.tril(b, diagonal=-1)
tensor([[ 0.0000,  0.0000,  0.0000,  0.0000,  0.0000,  0.0000],
        [ 0.4785,  0.0000,  0.0000,  0.0000,  0.0000,  0.0000],
        [ 1.1502,  3.2716,  0.0000,  0.0000,  0.0000,  0.0000],
        [-0.0614, -0.7344, -1.3164,  0.0000,  0.0000,  0.0000]])
```

torch.**tril_indices**(*row*, *column*, *offset=0*, *dtype=torch.long*, *device='cpu'*, *layout=torch.strided*) → Tensor

Returns the indices of the lower triangular part of a `row`-by- `column` matrix in a 2-by-N Tensor, where the first row contains row coordinates of all indices and the second row contains column coordinates. Indices are ordered based on rows and then columns.

The lower triangular part of the matrix is defined as the elements on and below the diagonal.

The argument `offset` controls which diagonal to consider. If `offset` = 0, all elements on and below the main diagonal are retained. A positive value includes just as many diagonals above the main diagonal, and similarly a negative value excludes just as many diagonals below the main diagonal. The main diagonal are the set of indices $\{(i, i)\}$ for $i \in [0, \min\{d_1, d_2\} - 1]$ where d_1, d_2 are the dimensions of the matrix.

NOTE: when running on cuda, row * col must be less than 2^{59} to prevent overflow during calculation.

Parameters

- **row** (`int`) – number of rows in the 2-D matrix.
- **column** (`int`) – number of columns in the 2-D matrix.
- **offset** (`int`) – diagonal offset from the main diagonal. Default: if not provided, 0.
- **dtype** (*torch.dtype*, optional) – the desired data type of returned tensor. Default: if `None`, `torch.long`.
- **device** (*torch.device*, optional) – the desired device of returned tensor. Default: if `None`, uses the current device for the default tensor type (see *torch.set_default_tensor_type()*). `device` will be the CPU for CPU tensor types and the current CUDA device for CUDA tensor types.
- **layout** (*torch.layout*, optional) – currently only support `torch.strided`.

Example::

```
>>> a = torch.tril_indices(3, 3)
>>> a
tensor([[0, 1, 1, 2, 2, 2],
        [0, 0, 1, 0, 1, 2]])
```

```
>>> a = torch.tril_indices(4, 3, -1)
>>> a
tensor([[1, 2, 2, 3, 3, 3],
        [0, 0, 1, 0, 1, 2]])
```

```
>>> a = torch.tril_indices(4, 3, 1)
>>> a
tensor([[0, 0, 1, 1, 1, 2, 2, 2, 3, 3, 3],
        [0, 1, 0, 1, 2, 0, 1, 2, 0, 1, 2]])
```

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

Returns the upper triangular part of a matrix (2-D tensor) or batch of matrices `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 *diagonal* controls which diagonal to consider. If *diagonal* = 0, all elements on and below the main diagonal are retained. A positive value excludes just as many diagonals above the main diagonal, and similarly a negative value includes just as many diagonals below the main diagonal. The main diagonal are the set of indices $\{(i, i)\}$ for $i \in [0, \min\{d_1, d_2\} - 1]$ where d_1, d_2 are the dimensions of the matrix.

Parameters

- **input** (Tensor) – the input tensor
- **diagonal** (*int, optional*) – the diagonal to consider
- **out** (Tensor, *optional*) – the output tensor

Example:

```
>>> a = torch.randn(3, 3)
>>> a
tensor([[ 0.2309,  0.5207,  2.0049],
        [ 0.2072, -1.0680,  0.6602],
        [ 0.3480, -0.5211, -0.4573]])
>>> torch.triu(a)
tensor([[ 0.2309,  0.5207,  2.0049],
        [ 0.0000, -1.0680,  0.6602],
        [ 0.0000,  0.0000, -0.4573]])
>>> torch.triu(a, diagonal=1)
tensor([[ 0.0000,  0.5207,  2.0049],
        [ 0.0000,  0.0000,  0.6602],
        [ 0.0000,  0.0000,  0.0000]])
>>> torch.triu(a, diagonal=-1)
tensor([[ 0.2309,  0.5207,  2.0049],
        [ 0.2072, -1.0680,  0.6602],
        [ 0.0000, -0.5211, -0.4573]])

>>> b = torch.randn(4, 6)
>>> b
tensor([[ 0.5876, -0.0794, -1.8373,  0.6654,  0.2604,  1.5235],
        [-0.2447,  0.9556, -1.2919,  1.3378, -0.1768, -1.0857],
        [ 0.4333,  0.3146,  0.6576, -1.0432,  0.9348, -0.4410],
        [-0.9888,  1.0679, -1.3337, -1.6556,  0.4798,  0.2830]])
>>> torch.triu(b, diagonal=1)
tensor([[ 0.0000, -0.0794, -1.8373,  0.6654,  0.2604,  1.5235],
        [ 0.0000,  0.0000, -1.2919,  1.3378, -0.1768, -1.0857],
        [ 0.0000,  0.0000,  0.0000, -1.0432,  0.9348, -0.4410],
        [ 0.0000,  0.0000,  0.0000,  0.0000,  0.4798,  0.2830]])
>>> torch.triu(b, diagonal=-1)
tensor([[ 0.5876, -0.0794, -1.8373,  0.6654,  0.2604,  1.5235],
        [-0.2447,  0.9556, -1.2919,  1.3378, -0.1768, -1.0857],
        [ 0.0000,  0.3146,  0.6576, -1.0432,  0.9348, -0.4410],
        [ 0.0000,  0.0000, -1.3337, -1.6556,  0.4798,  0.2830]])
```

torch.**triu_indices**(*row*, *column*, *offset=0*, *dtype=torch.long*, *device='cpu'*, *layout=torch.strided*) → Tensor

Returns the indices of the upper triangular part of a `row` by `column` matrix in a 2-by-N Tensor, where the first row contains row coordinates of all indices and the second row contains column coordinates. Indices are ordered based on rows and then columns.

The upper triangular part of the matrix is defined as the elements on and above the diagonal.

The argument `offset` controls which diagonal to consider. If `offset` = 0, all elements on and above the main diagonal are retained. A positive value excludes just as many diagonals above the main diagonal, and similarly a negative value includes just as many diagonals below the main diagonal. The main diagonal are the set of indices $\{(i, i)\}$ for $i \in [0, \min\{d_1, d_2\} - 1]$ where d_1, d_2 are the dimensions of the matrix.

NOTE: when running on cuda, row * col must be less than 2^{59} to prevent overflow during calculation.

Parameters

- **row** (`int`) – number of rows in the 2-D matrix.
- **column** (`int`) – number of columns in the 2-D matrix.
- **offset** (`int`) – diagonal offset from the main diagonal. Default: if not provided, 0.
- **dtype** (*torch.dtype*, optional) – the desired data type of returned tensor. Default: if `None`, `torch.long`.
- **device** (*torch.device*, optional) – the desired device of returned tensor. Default: if `None`, uses the current device for the default tensor type (see *torch.set_default_tensor_type()*). `device` will be the CPU for CPU tensor types and the current CUDA device for CUDA tensor types.
- **layout** (*torch.layout*, optional) – currently only support `torch.strided`.

Example::

```
>>> a = torch.triu_indices(3, 3)
>>> a
tensor([[0, 0, 0, 1, 1, 2],
        [0, 1, 2, 1, 2, 2]])
```

```
>>> a = torch.triu_indices(4, 3, -1)
>>> a
tensor([[0, 0, 0, 1, 1, 1, 2, 2, 3],
        [0, 1, 2, 0, 1, 2, 1, 2, 2]])
```

```
>>> a = torch.triu_indices(4, 3, 1)
>>> a
tensor([[0, 0, 1],
        [1, 2, 2]])
```

15.7.6 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 3-D tensors each containing the same number of matrices.

If `batch1` is a $(b \times n \times m)$ tensor, `batch2` is a $(b \times m \times p)$ tensor, `mat` must be *broadcastable* with a $(n \times p)$ tensor and `out` will be a $(n \times p)$ tensor.

$$out = \beta\ \text{mat} + \alpha\ (\sum_{i=0}^{b-1} \text{batch1}_i \mathbin{@} \text{batch2}_i)$$

For inputs of type *FloatTensor* or *DoubleTensor*, arguments `beta` and `alpha` must be real numbers, otherwise they should be integers.

Parameters

- **beta** (*Number, optional*) – multiplier for `mat` (β)
- **mat** (Tensor) – matrix to be added
- **alpha** (*Number, optional*) – multiplier for *batch1* @ *batch2* (α)
- **batch1** (Tensor) – the first batch of matrices to be multiplied
- **batch2** (Tensor) – the second batch of matrices to be multiplied
- **out** (Tensor, *optional*) – the 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)
tensor([[  6.6311,   0.0503,   6.9768, -12.0362,  -2.1653],
        [ -4.8185,  -1.4255,  -6.6760,   8.9453,   2.5743],
        [ -3.8202,   4.3691,   1.0943,  -1.1109,   5.4730]])
```

`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 $(n \times m)$ tensor, `mat2` is a $(m \times p)$ tensor, then `mat` must be *broadcastable* with a $(n \times p)$ tensor and `out` will be a $(n \times p)$ tensor.

`alpha` and `beta` are scaling factors on matrix-vector product between `mat1` and `mat2` and the added matrix `mat` respectively.

$$\text{out} = \beta\ \text{mat} + \alpha\ (\text{mat1}_i\ @\ \text{mat2}_i)$$

For inputs of type *FloatTensor* or *DoubleTensor*, arguments `beta` and `alpha` must be real numbers, otherwise they should be integers.

Parameters

- **beta** (*Number, optional*) – multiplier for `mat` (β)
- **mat** (Tensor) – matrix to be added
- **alpha** (*Number, optional*) – multiplier for $mat1@mat2$ (α)
- **mat1** (Tensor) – the first matrix to be multiplied
- **mat2** (Tensor) – the second matrix to be multiplied
- **out** (Tensor, *optional*) – the output tensor

Example:

```
>>> M = torch.randn(2, 3)
>>> mat1 = torch.randn(2, 3)
>>> mat2 = torch.randn(3, 3)
>>> torch.addmm(M, mat1, mat2)
tensor([[-4.8716,  1.4671, -1.3746],
        [ 0.7573, -3.9555, -2.8681]])
```

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 $(n \times m)$ tensor, `vec` is a 1-D tensor of size *m*, then *tensor* must be *broadcastable* with a 1-D tensor of size *n* and `out` will be 1-D tensor of size *n*.

`alpha` and `beta` are scaling factors on matrix-vector product between `mat` and `vec` and the added tensor *tensor* respectively.

$$\text{out} = \beta\ \text{tensor} + \alpha\ (\text{mat} \mathbin{@} \text{vec})$$

For inputs of type *FloatTensor* or *DoubleTensor*, arguments `beta` and `alpha` must be real numbers, otherwise they should be integers

Parameters

- **beta** (*Number, optional*) – multiplier for *tensor* (β)
- **tensor** (Tensor) – vector to be added
- **alpha** (*Number, optional*) – multiplier for $mat@vec$ (α)
- **mat** (Tensor) – matrix to be multiplied
- **vec** (Tensor) – vector to be multiplied
- **out** (Tensor, *optional*) – the output tensor

Example:

```
>>> M = torch.randn(2)
>>> mat = torch.randn(2, 3)
>>> vec = torch.randn(3)
>>> torch.addmv(M, mat, vec)
tensor([-0.3768, -5.5565])
```

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 scaling factors on the outer product between `vec1` and `vec2` and the added matrix `mat` respectively.

$$\text{out} = \beta\ \text{mat} + \alpha\ (\text{vec1} \otimes \text{vec2})$$

If `vec1` is a vector of size *n* and `vec2` is a vector of size *m*, then `mat` must be *broadcastable* with a matrix of size $(n \times m)$ and `out` will be a matrix of size $(n \times m)$.

For inputs of type *FloatTensor* or *DoubleTensor*, arguments `beta` and `alpha` must be real numbers, otherwise they should be integers

Parameters

- **beta** (*Number, optional*) – multiplier for `mat` (β)
- **mat** (Tensor) – matrix to be added
- **alpha** (*Number, optional*) – multiplier for vec1 $\otimes$ vec2 (α)
- **vec1** (Tensor) – the first vector of the outer product
- **vec2** (Tensor) – the second vector of the outer product
- **out** (Tensor, *optional*) – the output tensor

Example:

```
>>> vec1 = torch.arange(1., 4.)
>>> vec2 = torch.arange(1., 3.)
>>> M = torch.zeros(3, 2)
>>> torch.addr(M, vec1, vec2)
tensor([[ 1.,  2.],
        [ 2.,  4.],
        [ 3.,  6.]])
```

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 3-D tensors each containing the same number of matrices.

If `batch1` is a $(b \times n \times m)$ tensor, `batch2` is a $(b \times m \times p)$ tensor, then `mat` must be *broadcastable* with a $(b \times n \times p)$ tensor and `out` will be a $(b \times n \times p)$ tensor. Both `alpha` and `beta` mean the same as the scaling factors used in *torch.addbmm()*.

$$\text{out}_i = \beta\ \text{mat}_i + \alpha\ (\text{batch1}_i \mathbin{@} \text{batch2}_i)$$

For inputs of type *FloatTensor* or *DoubleTensor*, arguments `beta` and `alpha` must be real numbers, otherwise they should be integers.

Parameters

- **beta** (*Number, optional*) – multiplier for `mat` (β)
- **mat** (Tensor) – the tensor to be added
- **alpha** (*Number, optional*) – multiplier for batch1 @ batch2 (α)
- **batch1** (Tensor) – the first batch of matrices to be multiplied
- **batch2** (Tensor) – the second batch of matrices to be multiplied
- **out** (Tensor, *optional*) – the 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 3-D tensors each containing the same number of matrices.

If `batch1` is a $(b \times n \times m)$ tensor, `batch2` is a $(b \times m \times p)$ tensor, `out` will be a $(b \times n \times p)$ tensor.

$$\text{out}_i = \text{batch1}_i \mathbin{@} \text{batch2}_i$$

Note: This function does not *broadcast*. For broadcasting matrix products, see *torch.matmul()*.

Parameters

- **batch1** (Tensor) – the first batch of matrices to be multiplied

- **batch2** (Tensor) – the second batch of matrices to be multiplied
- **out** (Tensor, *optional*) – the 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.**chain_matmul**(**matrices*)

Returns the matrix product of the N 2-D tensors. This product is efficiently computed using the matrix chain order algorithm which selects the order in which incurs the lowest cost in terms of arithmetic operations ([CLRS]). Note that since this is a function to compute the product, N needs to be greater than or equal to 2; if equal to 2 then a trivial matrix-matrix product is returned. If N is 1, then this is a no-op - the original matrix is returned as is.

Parameters **matrices** (*Tensors...*) – a sequence of 2 or more 2-D tensors whose product is to be determined.

Returns if the i^{th} tensor was of dimensions $p_i \times p_{i+1}$, then the product would be of dimensions $p_1 \times p_{N+1}$.

Return type *Tensor*

Example:

```
>>> a = torch.randn(3, 4)
>>> b = torch.randn(4, 5)
>>> c = torch.randn(5, 6)
>>> d = torch.randn(6, 7)
>>> torch.chain_matmul(a, b, c, d)
tensor([[ -2.3375,  -3.9790,  -4.1119,  -6.6577,   9.5609, -11.5095,  -3.2614],
        [ 21.4038,   3.3378,  -8.4982,  -5.2457, -10.2561,  -2.4684,   2.7163],
        [ -0.9647,  -5.8917,  -2.3213,  -5.2284,  12.8615, -12.2816,  -2.5095]])
```

torch.**cholesky**(*A*, *upper=False*, *out=None*) → Tensor

Computes the Cholesky decomposition of a symmetric positive-definite matrix A or for batches of symmetric positive-definite matrices.

If `upper` is `True`, the returned matrix *U* is upper-triangular, and the decomposition has the form:

$$A = U^T U$$

If `upper` is `False`, the returned matrix *L* is lower-triangular, and the decomposition has the form:

$$A = LL^T$$

If `upper` is `True`, and `A` is a batch of symmetric positive-definite matrices, then the returned tensor will be composed of upper-triangular Cholesky factors of each of the individual matrices. Similarly, when `upper` is `False`, the returned tensor will be composed of lower-triangular Cholesky factors of each of the individual matrices.

Parameters

- **a** (Tensor) – the input tensor of size (*, n, n) where * is zero or more batch dimensions consisting of symmetric positive-definite matrices.

- **upper** (*bool, optional*) – flag that indicates whether to return a upper or lower triangular matrix. Default: `False`
- **out** (`Tensor`, *optional*) – the output matrix

Example:

```
>>> a = torch.randn(3, 3)
>>> a = torch.mm(a, a.t()) # make symmetric positive-definite
>>> l = torch.cholesky(a)
>>> a
tensor([[ 2.4112, -0.7486,  1.4551],
        [-0.7486,  1.3544,  0.1294],
        [ 1.4551,  0.1294,  1.6724]])
>>> l
tensor([[ 1.5528,  0.0000,  0.0000],
        [-0.4821,  1.0592,  0.0000],
        [ 0.9371,  0.5487,  0.7023]])
>>> torch.mm(l, l.t())
tensor([[ 2.4112, -0.7486,  1.4551],
        [-0.7486,  1.3544,  0.1294],
        [ 1.4551,  0.1294,  1.6724]])
>>> a = torch.randn(3, 2, 2)
>>> a = torch.matmul(a, a.transpose(-1, -2)) + 1e-03 # make symmetric positive-
↪definite
>>> l = torch.cholesky(a)
>>> z = torch.matmul(l, l.transpose(-1, -2))
>>> torch.max(torch.abs(z - a)) # Max non-zero
tensor(2.3842e-07)
```

`torch.`**`cholesky_solve`**(*b*, *u*, *upper=False*, *out=None*) → Tensor

Solves a linear system of equations with a positive semidefinite matrix to be inverted given its Cholesky factor matrix `u`.

If `upper` is `False`, `u` is and lower triangular and *c* is returned such that:

$$c = (uu^T)^{-1}b$$

If `upper` is `True` or not provided, `u` is upper triangular and *c* is returned such that:

$$c = (u^Tu)^{-1}b$$

torch.cholesky_solve(b, u) can take in 2D inputs *b, u* or inputs that are batches of 2D matrices. If the inputs are batches, then returns batched outputs *c*

Note: The `out` keyword only supports 2D matrix inputs, that is, *b, u* must be 2D matrices.

Parameters

- **b** (`Tensor`) – input matrix of size $(*, m, k)$, where $*$ is zero or more batch dimensions
- **u** (`Tensor`) – input matrix of size $(*, m, m)$, where $*$ is zero of more batch dimensions composed of upper or lower triangular Cholesky factor
- **upper** (*bool, optional*) – whether to consider the Cholesky factor as a lower or upper triangular matrix. Default: `False`.
- **out** (`Tensor`, *optional*) – the output tensor for *c*

Example:

```
>>> a = torch.randn(3, 3)
>>> a = torch.mm(a, a.t()) # make symmetric positive definite
>>> u = torch.cholesky(a)
>>> a
tensor([[ 0.7747, -1.9549,  1.3086],
        [-1.9549,  6.7546, -5.4114],
        [ 1.3086, -5.4114,  4.8733]])
>>> b = torch.randn(3, 2)
>>> b
tensor([[-0.6355,  0.9891],
        [ 0.1974,  1.4706],
        [-0.4115, -0.6225]])
>>> torch.cholesky_solve(b, u)
tensor([[ -8.1625,  19.6097],
        [ -5.8398,  14.2387],
        [ -4.3771,  10.4173]])
>>> torch.mm(a.inverse(), b)
tensor([[ -8.1626,  19.6097],
        [ -5.8398,  14.2387],
        [ -4.3771,  10.4173]])
```

`torch.`**`dot`**(*tensor1*, *tensor2*) → Tensor

Computes the dot product (inner product) of two tensors.

Note: This function does not *broadcast*.

Example:

```
>>> torch.dot(torch.tensor([2, 3]), torch.tensor([2, 1]))
tensor(7)
```

`torch.`**`eig`**(*a*, *eigenvectors=False*, *out=None)* -> *(Tensor*, *Tensor*)

Computes the eigenvalues and eigenvectors of a real square matrix.

Note: Since eigenvalues and eigenvectors might be complex, backward pass is supported only

---

for *torch.symeig()*

Parameters

- **a** (Tensor) – the square matrix of shape $(n \times n)$ 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*) – the output tensors

Returns

A namedtuple (eigenvalues, eigenvectors) containing

- **eigenvalues** (*Tensor*): Shape $(n \times 2)$. Each row is an eigenvalue of `a`, where the first element is the real part and the second element is the imaginary part. The eigenvalues are not necessarily ordered.

- **eigenvectors** (*Tensor*): If `eigenvectors=False`, its an empty tensor. Otherwise, this tensor of shape $(n \times n)$ can be used to compute normalized (unit length) eigenvectors of corresponding eigenvalues as follows. If the corresponding *eigenvalues[j]* is a real number, column *eigenvectors[:, j]* is the eigenvector corresponding to *eigenvalues[j]*. If the corresponding *eigenvalues[j]* and *eigenvalues[j + 1]* form a complex conjugate pair, then the true eigenvectors can be computed as true eigenvector$[j] = eigenvectors[:, j] + i \times eigenvectors[:, j + 1]$, true eigenvector$[j + 1] = eigenvectors[:, j] - i \times eigenvectors[:, j + 1]$.

Return type (*Tensor*, *Tensor*)

`torch.`**`geqrf`**(*input*, *out=None*) -> (*Tensor*, *Tensor*)

This is a low-level function for calling LAPACK directly. This function returns a namedtuple (a, tau) as defined in LAPACK documentation for geqrf .

Youll 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 geqrf for further details.

Parameters

- **`input`** (*`Tensor`*) – the input matrix
- **`out`** (*`tuple, optional`*) – the output 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 $(n \times m)$.

Note: This function does not *broadcast*.

Parameters

- **`vec1`** (*`Tensor`*) – 1-D input vector
- **`vec2`** (*`Tensor`*) – 1-D input vector
- **`out`** (*`Tensor, optional`*) – optional output matrix

Example:

```
>>> v1 = torch.arange(1., 5.)
>>> v2 = torch.arange(1., 4.)
>>> torch.ger(v1, v2)
tensor([[  1.,   2.,   3.],
        [  2.,   4.,   6.],
        [  3.,   6.,   9.],
        [  4.,   8.,  12.]])
```

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

Takes the inverse of the square matrix `input`. `input` can be batches of 2D square tensors, in which case this function would return a tensor composed of individual inverses.

Note: Irrespective of the original strides, the returned tensors will be transposed, i.e. with strides like *input.contiguous().transpose(-2, -1).strides()*

Parameters

- **input** (Tensor) – the input tensor of size (*, n, n) where * is zero or more batch dimensions
- **out** (Tensor, *optional*) – the optional output tensor

Example:

```
>>> x = torch.rand(4, 4)
>>> y = torch.inverse(x)
>>> z = torch.mm(x, y)
>>> z
tensor([[ 1.0000, -0.0000, -0.0000,  0.0000],
        [ 0.0000,  1.0000,  0.0000,  0.0000],
        [ 0.0000,  0.0000,  1.0000,  0.0000],
        [ 0.0000, -0.0000, -0.0000,  1.0000]])
>>> torch.max(torch.abs(z - torch.eye(4))) # Max non-zero
tensor(1.1921e-07)
>>> # Batched inverse example
>>> x = torch.randn(2, 3, 4, 4)
>>> y = torch.inverse(x)
>>> z = torch.matmul(x, y)
>>> torch.max(torch.abs(z - torch.eye(4).expand_as(x))) # Max non-zero
tensor(1.9073e-06)
```

torch.**det**(*A*) → Tensor

Calculates determinant of a 2D square tensor.

Note: Backward through *det()* internally uses SVD results when A is not invertible. In this case, double backward through *det()* will be unstable in when A doesnt have distinct singular values. See *svd()* for details.

Parameters **A** (Tensor) – The input 2D square tensor

Example:

```
>>> A = torch.randn(3, 3)
>>> torch.det(A)
tensor(3.7641)
```

torch.**logdet**(*A*) → Tensor

Calculates log determinant of a 2D square tensor.

Note: Result is -inf if A has zero log determinant, and is nan if A has negative determinant.

Note: Backward through *logdet()* internally uses SVD results when A is not invertible. In this case, double backward through *logdet()* will be unstable in when A doesnt have distinct singular values. See *svd()* for

details.

Parameters **A** (Tensor) – The input 2D square tensor

Example:

```
>>> A = torch.randn(3, 3)
>>> torch.det(A)
tensor(0.2611)
>>> torch.logdet(A)
tensor(-1.3430)
```

torch.**slogdet**(*A*) -> (*Tensor*, *Tensor*)

Calculates the sign and log value of a 2D square tensors determinant.

Note: If A has zero determinant, this returns (0, -inf).

Note: Backward through *slogdet()* internally uses SVD results when A is not invertible. In this case, double backward through *slogdet()* will be unstable in when A doesnt have distinct singular values. See *svd()* for details.

Parameters **A** (Tensor) – The input 2D square tensor

Returns A namedtuple (sign, logabsdet) containing the sign of the determinant, and the log value of the absolute determinant.

Example:

```
>>> A = torch.randn(3, 3)
>>> A
tensor([[ 0.0032, -0.2239, -1.1219],
        [-0.6690,  0.1161,  0.4053],
        [-1.6218, -0.9273, -0.0082]])
>>> torch.det(A)
tensor(-0.7576)
>>> torch.logdet(A)
tensor(nan)
>>> torch.slogdet(A)
torch.return_types.slogdet(sign=tensor(-1.), logabsdet=tensor(-0.2776))
```

torch.**lu**(*A*, *pivot=True*, *get_infos=False*, *out=None*)

Computes the LU factorization of a square matrix or batches of square matrices A. Returns a tuple containing the LU factorization and pivots of A. Pivoting is done if pivot is set to True.

Note: The pivots returned by the function are 1-indexed. If pivot is False, then the returned pivots is a tensor filled with zeros of the appropriate size.

Note: LU factorization with pivot = False is not available for CPU, and attempting to do so will throw an error. However, LU factorization with pivot = False is available for CUDA.

Note: This function does not check if the factorization was successful or not if `get_infos` is `True` since the status of the factorization is present in the third element of the return tuple.

Parameters

- **A** (`Tensor`) – the tensor to factor of size $(*, m, m)$
- **pivot** (*bool, optional*) – controls whether pivoting is done. Default: `True`
- **get_infos** (*bool, optional*) – if set to `True`, returns an info IntTensor. Default: `False`
- **out** (*tuple, optional*) – optional output tuple. If `get_infos` is `True`, then the elements in the tuple are Tensor, IntTensor, and IntTensor. If `get_infos` is `False`, then the elements in the tuple are Tensor, IntTensor. Default: `None`

Returns

A tuple of tensors containing

- **factorization** (*Tensor*): the factorization of size $(*, m, m)$
- **pivots** (*IntTensor*): the pivots of size $(*, m)$
- **infos** (*IntTensor*, *optional*): if `get_infos` is `True`, this is a tensor of size $(*)$ where non-zero values indicate whether factorization for the matrix or each minibatch has succeeded or failed

Return type (*Tensor*, IntTensor, IntTensor (optional))

Example:

```
>>> A = torch.randn(2, 3, 3)
>>> A_LU, pivots = torch.lu(A)
>>> A_LU
tensor([[[ 1.3506,  2.5558, -0.0816],
         [ 0.1684,  1.1551,  0.1940],
         [ 0.1193,  0.6189, -0.5497]],

        [[ 0.4526,  1.2526, -0.3285],
         [-0.7988,  0.7175, -0.9701],
         [ 0.2634, -0.9255, -0.3459]]])
>>> pivots
tensor([[ 3,  3,  3],
        [ 3,  3,  3]], dtype=torch.int32)
>>> A_LU, pivots, info = torch.lu(A, get_infos=True)
>>> if info.nonzero().size(0) == 0:
...   print('LU factorization succeeded for all samples!')
LU factorization succeeded for all samples!
```

`torch.`**`lu_unpack`**(*LU_data*, *LU_pivots*, *unpack_data=True*, *unpack_pivots=True*)

Unpacks the data and pivots from a LU factorization of a tensor.

Returns a tuple of tensors as `(the pivots, the L tensor, the U tensor)`.

Parameters

- **LU_data** (`Tensor`) – the packed LU factorization data
- **LU_pivots** (`Tensor`) – the packed LU factorization pivots

- **unpack_data** (*bool*) – flag indicating if the data should be unpacked
- **unpack_pivots** (*bool*) – flag indicating if the pivots should be unpacked

Example:

```
>>> A = torch.randn(2, 3, 3)
>>> A_LU, pivots = A.lu()
>>> P, A_L, A_U = torch.lu_unpack(A_LU, pivots)
>>>
>>> # can recover A from factorization
>>> A_ = torch.bmm(P, torch.bmm(A_L, A_U))
```

torch.**matmul**(*tensor1*, *tensor2*, *out=None*) → Tensor

Matrix product of two tensors.

The behavior depends on the dimensionality of the tensors as follows:

- If both tensors are 1-dimensional, the dot product (scalar) is returned.
- If both arguments are 2-dimensional, the matrix-matrix product is returned.
- If the first argument is 1-dimensional and the second argument is 2-dimensional, a 1 is prepended to its dimension for the purpose of the matrix multiply. After the matrix multiply, the prepended dimension is removed.
- If the first argument is 2-dimensional and the second argument is 1-dimensional, the matrix-vector product is returned.
- If both arguments are at least 1-dimensional and at least one argument is N-dimensional (where N > 2), then a batched matrix multiply is returned. If the first argument is 1-dimensional, a 1 is prepended to its dimension for the purpose of the batched matrix multiply and removed after. If the second argument is 1-dimensional, a 1 is appended to its dimension for the purpose of the batched matrix multiple and removed after. The non-matrix (i.e. batch) dimensions are *broadcasted* (and thus must be broadcastable). For example, if `tensor1` is a $(j \times 1 \times n \times m)$ tensor and `tensor2` is a $(k \times m \times p)$ tensor, `out` will be an $(j \times k \times n \times p)$ tensor.

Note: The 1-dimensional dot product version of this function does not support an `out` parameter.

Parameters

- **tensor1** (Tensor) – the first tensor to be multiplied
- **tensor2** (Tensor) – the second tensor to be multiplied
- **out** (Tensor, *optional*) – the output tensor

Example:

```
>>> # vector x vector
>>> tensor1 = torch.randn(3)
>>> tensor2 = torch.randn(3)
>>> torch.matmul(tensor1, tensor2).size()
torch.Size([])
>>> # matrix x vector
>>> tensor1 = torch.randn(3, 4)
>>> tensor2 = torch.randn(4)
>>> torch.matmul(tensor1, tensor2).size()
torch.Size([3])
```

(continues on next page)

(continued from previous page)

```
>>> # batched matrix x broadcasted vector
>>> tensor1 = torch.randn(10, 3, 4)
>>> tensor2 = torch.randn(4)
>>> torch.matmul(tensor1, tensor2).size()
torch.Size([10, 3])
>>> # batched matrix x batched matrix
>>> tensor1 = torch.randn(10, 3, 4)
>>> tensor2 = torch.randn(10, 4, 5)
>>> torch.matmul(tensor1, tensor2).size()
torch.Size([10, 3, 5])
>>> # batched matrix x broadcasted matrix
>>> tensor1 = torch.randn(10, 3, 4)
>>> tensor2 = torch.randn(4, 5)
>>> torch.matmul(tensor1, tensor2).size()
torch.Size([10, 3, 5])
```

torch.**matrix_power**(*input*, *n*) → Tensor

Returns the matrix raised to the power `n` for square matrices. For batch of matrices, each individual matrix is raised to the power `n`.

If `n` is negative, then the inverse of the matrix (if invertible) is raised to the power `n`. For a batch of matrices, the batched inverse (if invertible) is raised to the power `n`. If `n` is 0, then an identity matrix is returned.

Parameters

- **input** (Tensor) – the input tensor
- **n** (*int*) – the power to raise the matrix to

Example:

```
>>> a = torch.randn(2, 2, 2)
>>> a
tensor([[[-1.9975, -1.9610],
         [ 0.9592, -2.3364]],

        [[-1.2534, -1.3429],
         [ 0.4153, -1.4664]]])
>>> torch.matrix_power(a, 3)
tensor([[[  3.9392, -23.9916],
         [ 11.7357,  -0.2070]],

        [[  0.2468,  -6.7168],
         [  2.0774,  -0.8187]]])
```

torch.**matrix_rank**(*input*, *tol=None*, *bool symmetric=False*) → Tensor

Returns the numerical rank of a 2-D tensor. The method to compute the matrix rank is done using SVD by default. If `symmetric` is `True`, then `input` is assumed to be symmetric, and the computation of the rank is done by obtaining the eigenvalues.

`tol` is the threshold below which the singular values (or the eigenvalues when `symmetric` is `True`) are considered to be 0. If `tol` is not specified, `tol` is set to `S.max() * max(S.size()) * eps` where *S* is the singular values (or the eigenvalues when `symmetric` is `True`), and `eps` is the epsilon value for the datatype of `input`.

Parameters

- **input** (Tensor) – the input 2-D tensor
- **tol** (*float, optional*) – the tolerance value. Default: `None`

- **symmetric** (*bool, optional*) – indicates whether `input` is symmetric. Default: `False`

Example:

```
>>> a = torch.eye(10)
>>> torch.matrix_rank(a)
tensor(10)
>>> b = torch.eye(10)
>>> b[0, 0] = 0
>>> torch.matrix_rank(b)
tensor(9)
```

`torch.mm`(*mat1*, *mat2*, *out=None*) → Tensor

Performs a matrix multiplication of the matrices `mat1` and `mat2`.

If `mat1` is a $(n \times m)$ tensor, `mat2` is a $(m \times p)$ tensor, `out` will be a $(n \times p)$ tensor.

Note: This function does not *broadcast*. For broadcasting matrix products, see *torch.matmul()*.

Parameters

- **mat1** (*Tensor*) – the first matrix to be multiplied
- **mat2** (*Tensor*) – the second matrix to be multiplied
- **out** (*Tensor, optional*) – the output tensor

Example:

```
>>> mat1 = torch.randn(2, 3)
>>> mat2 = torch.randn(3, 3)
>>> torch.mm(mat1, mat2)
tensor([[ 0.4851,  0.5037, -0.3633],
        [-0.0760, -3.6705,  2.4784]])
```

`torch.mv`(*mat*, *vec*, *out=None*) → Tensor

Performs a matrix-vector product of the matrix `mat` and the vector `vec`.

If `mat` is a $(n \times m)$ tensor, `vec` is a 1-D tensor of size m, `out` will be 1-D of size n.

Note: This function does not *broadcast*.

Parameters

- **mat** (*Tensor*) – matrix to be multiplied
- **vec** (*Tensor*) – vector to be multiplied
- **out** (*Tensor, optional*) – the output tensor

Example:

```
>>> mat = torch.randn(2, 3)
>>> vec = torch.randn(3)
>>> torch.mv(mat, vec)
tensor([ 1.0404, -0.6361])
```

torch.**orgqr**(*a*, *tau*) → Tensor

Computes the orthogonal matrix *Q* of a QR factorization, from the *(a, tau)* tuple returned by `torch.geqrf()`.

This directly calls the underlying LAPACK function *?orgqr*. See LAPACK documentation for orgqr for further details.

Parameters

- **a** (Tensor) – the *a* from `torch.geqrf()`.
- **tau** (Tensor) – the *tau* from `torch.geqrf()`.

torch.**ormqr**(*a*, *tau*, *mat*, *left=True*, *transpose=False*) → Tensor

Multiplies *mat* by the orthogonal *Q* matrix of the QR factorization formed by `torch.geqrf()` that is represented by *(a, tau)*.

This directly calls the underlying LAPACK function *?ormqr*. See LAPACK documentation for ormqr for further details.

Parameters

- **a** (Tensor) – the *a* from `torch.geqrf()`.
- **tau** (Tensor) – the *tau* from `torch.geqrf()`.
- **mat** (Tensor) – the matrix to be multiplied.

torch.**pinverse**(*input*, *rcond=1e-15*) → Tensor

Calculates the pseudo-inverse (also known as the Moore-Penrose inverse) of a 2D tensor. Please look at Moore-Penrose inverse for more details

---

Note: This method is implemented using the Singular Value Decomposition.

Note: The pseudo-inverse is not necessarily a continuous function in the elements of the matrix [1]. Therefore, derivatives are not always existent, and exist for a constant rank only [2]. However, this method is backprop-able due to the implementation by using SVD results, and could be unstable. Double-backward will also be unstable due to the usage of SVD internally. See `svd()` for more details.

Parameters

- **input** (Tensor) – The input 2D tensor of dimensions $m \times n$
- **rcond** (*float*) – A floating point value to determine the cutoff for small singular values. Default: 1e-15

Returns The pseudo-inverse of `input` of dimensions $n \times m$

Example:

```
>>> input = torch.randn(3, 5)
>>> input
tensor([[ 0.5495,  0.0979, -1.4092, -0.1128,  0.4132],
        [-1.1143, -0.3662,  0.3042,  1.6374, -0.9294],
        [-0.3269, -0.5745, -0.0382, -0.5922, -0.6759]])
>>> torch.pinverse(input)
tensor([[ 0.0600, -0.1933, -0.2090],
        [-0.0903, -0.0817, -0.4752],
        [-0.7124, -0.1631, -0.2272],
```

(continues on next page)

(continued from previous page)

```
        [ 0.1356,  0.3933, -0.5023],
        [-0.0308, -0.1725, -0.5216]])
```

torch.**qr**(*input*, *out=None*) -> (*Tensor*, *Tensor*)

Computes the QR decomposition of a matrix `input`, and returns a namedtuple (Q, R) of matrices such that input $= QR$, with Q being an orthogonal matrix and R being an upper triangular matrix.

This returns the thin (reduced) QR factorization.

Note: precision may be lost if the magnitudes of the elements of `input` are large

---

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

Note: Irrespective of the original strides, the returned matrix Q will be transposed, i.e. with strides *(1, m)* instead of *(m, 1)*.

Parameters

- **input** (*Tensor*) – the input 2-D tensor
- **out** (*tuple, optional*) – 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
tensor([[-0.8571,  0.3943,  0.3314],
        [-0.4286, -0.9029, -0.0343],
        [ 0.2857, -0.1714,  0.9429]])
>>> r
tensor([[ -14.0000,  -21.0000,   14.0000],
        [   0.0000, -175.0000,   70.0000],
        [   0.0000,    0.0000,  -35.0000]])
>>> torch.mm(q, r).round()
tensor([[  12.,  -51.,    4.],
        [   6.,  167.,  -68.],
        [  -4.,   24.,  -41.]])
>>> torch.mm(q.t(), q).round()
tensor([[ 1.,  0.,  0.],
        [ 0.,  1., -0.],
        [ 0., -0.,  1.]])
```

torch.**solve**(*B*, *A*, *out=None*) -> (*Tensor*, *Tensor*)

This function returns the solution to the system of linear equations represented by $AX = B$ and the LU factorization of A, in order as a namedtuple *solution, LU*.

LU contains *L* and *U* factors for LU factorization of *A*.

torch.solve(B, A) can take in 2D inputs *B, A* or inputs that are batches of 2D matrices. If the inputs are batches, then returns batched outputs *solution, LU*.

Note: Irrespective of the original strides, the returned matrices *solution* and *LU* will be transposed, i.e. with strides like *B.contiguous().transpose(-1, -2).strides()* and *A.contiguous().transpose(-1, -2).strides()* respectively.

Parameters

- **B** (Tensor) – input matrix of size $(*, m, k)$, where $*$ is zero or more batch dimensions.
- **A** (Tensor) – input square matrix of size $(*, m, m)$, where $*$ is zero or more batch dimensions.
- **out** ((Tensor, Tensor), optional) – optional output tuple.

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.solve(B, A)
>>> torch.dist(B, torch.mm(A, X))
tensor(1.00000e-06 *
       7.0977)

>>> # Batched solver example
>>> A = torch.randn(2, 3, 1, 4, 4)
>>> B = torch.randn(2, 3, 1, 4, 6)
>>> X, LU = torch.solve(B, A)
>>> torch.dist(B, A.matmul(X))
tensor(1.00000e-06 *
   3.6386)
```

torch.**svd**(*input*, *some=True*, *compute_uv=True*, *out=None*) -> (*Tensor*, *Tensor*, *Tensor*)

`svd(A)` returns a namedtuple `(U, S, V)` which the singular value decomposition of a input real matrix *A* of size *(n x m)* such that $A = USV^T$.

U is of shape $(n \times n)$.

S is a diagonal matrix of shape $(n \times m)$, represented as a vector of size $\min(n, m)$ containing the non-negative diagonal entries.

V is of shape $(m \times m)$.

If `some` is `True` (default), the returned *U* and *V* matrices will contain only $min(n, m)$ orthonormal columns.

If `compute_uv` is `False`, the returned *U* and *V* matrices will be zero matrices of shape $(n \times n)$ and $(m \times m)$ respectively. `some` will be ignored here.

Note: The implementation of SVD on CPU uses the LAPACK routine *?gesdd* (a divide-and-conquer algorithm) instead of *?gesvd* for speed. Analogously, the SVD on GPU uses the MAGMA routine *gesdd* as well.

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

instead of *(n, 1)*.

Note: Extra care needs to be taken when backward through *U* and *V* outputs. Such operation is really only stable when `input` is full rank with all distinct singular values. Otherwise, `NaN` can appear as the gradients are not properly defined. Also, notice that double backward will usually do an additional backward through *U* and *V* even if the original backward is only on *S*.

Note: When `some = False`, the gradients on `U[:, min(n, m):]` and `V[:, min(n, m):]` will be ignored in backward as those vectors can be arbitrary bases of the subspaces.

Note: When `compute_uv = False`, backward cannot be performed since `U` and `V` from the forward pass is required for the backward operation.

Parameters

- **input** (*Tensor*) – the input 2-D tensor
- **some** (*bool, optional*) – controls the shape of returned *U* and *V*
- **out** (*tuple, optional*) – the output tuple of tensors

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()

>>> torch.svd(a).__class__
<class 'torch.return_types.svd'>
>>> u, s, v = torch.svd(a)
>>> u
tensor([[-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]])
>>> s
tensor([ 27.4687,  22.6432,   8.5584,   5.9857,   2.0149])
>>> v
tensor([[-0.2514,  0.8148, -0.2606,  0.3967, -0.2180],
        [-0.3968,  0.3587,  0.7008, -0.4507,  0.1402],
        [-0.6922, -0.2489, -0.2208,  0.2513,  0.5891],
        [-0.3662, -0.3686,  0.3859,  0.4342, -0.6265],
        [-0.4076, -0.0980, -0.4933, -0.6227, -0.4396]])
>>> torch.dist(a, torch.mm(torch.mm(u, torch.diag(s)), v.t()))
tensor(1.00000e-06 *
       9.3738)
```

`torch.`**`symeig`**(*input*, *eigenvectors=False*, *upper=True*, *out=None*) -> (*Tensor*, *Tensor*)

This function returns eigenvalues and eigenvectors of a real symmetric matrix `input`, represented by a named-tuple (eigenvalues, eigenvectors).

`input` and V are $(m \times m)$ matrices and e is a m dimensional vector.

This function calculates all eigenvalues (and vectors) of `input` such that input $= V\text{diag}(e)V^T$.

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)*.

Note: Extra care needs to be taken when backward through outputs. Such operation is really only stable when all eigenvalues are distinct. Otherwise, `NaN` can appear as the gradients are not properly defined.

Parameters

- **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 output tuple of (Tensor, Tensor)

Returns

A namedtuple (eigenvalues, eigenvectors) containing

- **eigenvalues** (*Tensor*): Shape (m). Each element is an eigenvalue of `input`, The eigenvalues are in ascending order.
- **eigenvectors** (*Tensor*): Shape $(m \times m)$. If `eigenvectors=False`, its a tensor filled with zeros. Otherwise, this tensor contains the orthonormal eigenvectors of the `input`.

Return type (*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
tensor([-11.0656,  -6.2287,   0.8640,   8.8655,  16.0948])
>>> v
tensor([[-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],
```

(continues on next page)

(continued from previous page)

```
        [-0.0036, -0.4467,  0.4553,  0.6204, -0.4564],
        [-0.8041,  0.4480,  0.1725,  0.3108,  0.1622]])
```

torch.**triangular_solve**(*b*, *A*, *upper=True*, *transpose=False*, *unitriangular=False) -> (Tensor*, *Tensor)*

Solves a system of equations with a triangular coefficient matrix A and multiple right-hand sides `b`.

In particular, solves $AX = b$ and assumes A is upper-triangular with the default keyword arguments.

torch.triangular_solve(b, A) can take in 2D inputs *b, A* or inputs that are batches of 2D matrices. If the inputs are batches, then returns batched outputs *X*

Note: The `out` keyword only supports 2D matrix inputs, that is, *b, A* must be 2D matrices.

Parameters

- **A** (`Tensor`) – the input triangular coefficient matrix of size $(*, m, m)$ where $*$ is zero or more batch dimensions
- **b** (`Tensor`) – multiple right-hand sides of size $(*, m, k)$ where $*$ is zero of more batch dimensions
- **upper** (*bool, optional*) – whether to solve the upper-triangular system of equations (default) or the lower-triangular system of equations. Default: `True`.
- **transpose** (*bool, optional*) – whether A should be transposed before being sent into the solver. Default: `False`.
- **unitriangular** (*bool, optional*) – whether A is unit triangular. If True, the diagonal elements of A are assumed to be 1 and not referenced from A. Default: `False`.

Returns A tuple (X, M) where M is a clone of A and X is the solution to $AX = b$ (or whatever variant of the system of equations, depending on the keyword arguments.)

Examples:

```
>>> A = torch.randn(2, 2).triu()
>>> A
tensor([[ 1.1527, -1.0753],
        [ 0.0000,  0.7986]])
>>> b = torch.randn(2, 3)
>>> b
tensor([[-0.0210,  2.3513, -1.5492],
        [ 1.5429,  0.7403, -1.0243]])
>>> torch.triangular_solve(b, A)
(tensor([[ 1.7840,  2.9045, -2.5405],
        [ 1.9319,  0.9269, -1.2826]]), tensor([[ 1.1527, -1.0753],
        [ 0.0000,  0.7986]]))
```

15.8 Utilities

torch.**compiled_with_cxx11_abi**()

Returns whether PyTorch was built with _GLIBCXX_USE_CXX11_ABI=1

CHAPTER

SIXTEEN

TORCH.NN

16.1 Parameters

class torch.nn.**Parameter**

A kind of Tensor that is to be considered a module parameter.

Parameters are *Tensor* subclasses, that have a very special property when used with *Module* s - when theyre assigned as Module attributes they are automatically added to the list of its parameters, and will appear e.g. in *parameters()* iterator. Assigning a Tensor doesnt 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.

Parameters

- **data** (Tensor) – parameter tensor.
- **requires_grad** (*bool, optional*) – if the parameter requires gradient. See *Excluding subgraphs from backward* for more details. Default: *True*

16.2 Containers

16.2.1 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 *to()*, 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.

Parameters

- **name** (*string*) – name of the child module. The child module can be accessed from this module using the given name
- **module** (Module) – child module to be added to the module.

apply(*fn*)
Applies `fn` recursively to every submodule (as returned by `.children()`) as well as self. Typical use includes initializing the parameters of a model (see also torch-nn-init).

Parameters **fn** (*Module* -> None) – function to be applied to each submodule

Returns self

Return type *Module*

Example:

```
>>> def init_weights(m):
        print(m)
        if type(m) == nn.Linear:
            m.weight.data.fill_(1.0)
            print(m.weight)

>>> net = nn.Sequential(nn.Linear(2, 2), nn.Linear(2, 2))
>>> net.apply(init_weights)
Linear(in_features=2, out_features=2, bias=True)
Parameter containing:
tensor([[ 1.,  1.],
        [ 1.,  1.]])
Linear(in_features=2, out_features=2, bias=True)
Parameter containing:
tensor([[ 1.,  1.],
        [ 1.,  1.]])
Sequential(
  (0): Linear(in_features=2, out_features=2, bias=True)
  (1): Linear(in_features=2, out_features=2, bias=True)
)
Sequential(
  (0): Linear(in_features=2, out_features=2, bias=True)
  (1): Linear(in_features=2, out_features=2, bias=True)
)
```

buffers(*recurse=True*)
Returns an iterator over module buffers.

Parameters **recurse** (*bool*) – if True, then yields buffers of this module and all submodules. Otherwise, yields only buffers that are direct members of this module.

Yields *torch.Tensor* – module buffer

Example:

```
>>> for buf in model.buffers():
>>>     print(type(buf.data), buf.size())
<class 'torch.FloatTensor'> (20L,)
<class 'torch.FloatTensor'> (20L, 1L, 5L, 5L)
```

children()
Returns an iterator over immediate children modules.

Yields *Module* – a child module

cpu()
Moves all model parameters and buffers to the CPU.

Returns self

Return type *Module*

cuda(*device=None*)
Moves all model parameters and buffers to the GPU.

This also makes associated parameters and buffers different objects. So it should be called before constructing optimizer if the module will live on GPU while being optimized.

Parameters **device** (*int, optional*) – if specified, all parameters will be copied to that device

Returns self

Return type *Module*

double()
Casts all floating point parameters and buffers to `double` datatype.

Returns self

Return type *Module*

dump_patches = False
This allows better BC support for *load_state_dict()*. In *state_dict()*, the version number will be saved as in the attribute *_metadata* of the returned state dict, and thus pickled. *_metadata* is a dictionary with keys that follow the naming convention of state dict. See `_load_from_state_dict` on how to use this information in loading.

If new parameters/buffers are added/removed from a module, this number shall be bumped, and the modules *_load_from_state_dict* method can compare the version number and do appropriate changes if the state dict is from before the change.

eval()
Sets the module in evaluation mode.

This has any effect only on certain modules. See documentations of particular modules for details of their behaviors in training/evaluation mode, if they are affected, e.g. *Dropout*, `BatchNorm`, etc.

extra_repr()
Set the extra representation of the module

To print customized extra information, you should reimplement this method in your own modules. Both single-line and multi-line strings are acceptable.

float()
Casts all floating point parameters and buffers to float datatype.

Returns self

Return type *Module*

forward(**input*)

Defines the computation performed at every call.

Should be overridden by all subclasses.

Note: Although the recipe for forward pass needs to be defined within this function, one should call the `Module` instance afterwards instead of this since the former takes care of running the registered hooks while the latter silently ignores them.

---

half()

Casts all floating point parameters and buffers to `half` datatype.

Returns self

Return type *Module*

load_state_dict(*state_dict*, *strict=True*)

Copies parameters and buffers from `state_dict` into this module and its descendants. If `strict` is `True`, then the keys of `state_dict` must exactly match the keys returned by this modules `state_dict()` function.

Parameters

- **state_dict** (*dict*) – a dict containing parameters and persistent buffers.
- **strict** (*bool, optional*) – whether to strictly enforce that the keys in `state_dict` match the keys returned by this modules `state_dict()` function. Default: `True`

modules()

Returns an iterator over all modules in the network.

Yields *Module* – a module in the network

Note: Duplicate modules are returned only once. In the following example, `l` will be returned only once.

Example:

```
>>> l = nn.Linear(2, 2)
>>> net = nn.Sequential(l, l)
>>> for idx, m in enumerate(net.modules()):
        print(idx, '->', m)

0 -> Sequential(
  (0): Linear(in_features=2, out_features=2, bias=True)
  (1): Linear(in_features=2, out_features=2, bias=True)
)
1 -> Linear(in_features=2, out_features=2, bias=True)
```

named_buffers(*prefix=''*, *recurse=True*)

Returns an iterator over module buffers, yielding both the name of the buffer as well as the buffer itself.

Parameters

- **prefix** (*str*) – prefix to prepend to all buffer names.

- **recurse** (*bool*) – if True, then yields buffers of this module and all submodules. Otherwise, yields only buffers that are direct members of this module.

Yields *(string, torch.Tensor)* – Tuple containing the name and buffer

Example:

```
>>> for name, buf in self.named_buffers():
>>>     if name in ['running_var']:
>>>         print(buf.size())
```

named_children()

Returns an iterator over immediate children modules, yielding both the name of the module as well as the module itself.

Yields *(string, Module)* – Tuple containing a name and child module

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.

Yields *(string, Module)* – Tuple of name and module

Note: Duplicate modules are returned only once. In the following example, `l` will be returned only once.

Example:

```
>>> 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(in_features=2, out_features=2, bias=True)
  (1): Linear(in_features=2, out_features=2, bias=True)
))
1 -> ('0', Linear(in_features=2, out_features=2, bias=True))
```

named_parameters(*prefix=''*, *recurse=True*)

Returns an iterator over module parameters, yielding both the name of the parameter as well as the parameter itself.

Parameters

- **prefix** (*str*) – prefix to prepend to all parameter names.
- **recurse** (*bool*) – if True, then yields parameters of this module and all submodules. Otherwise, yields only parameters that are direct members of this module.

Yields *(string, Parameter)* – Tuple containing the name and parameter

Example:

```
>>> for name, param in self.named_parameters():
>>>     if name in ['bias']:
>>>         print(param.size())
```

parameters(*recurse=True*)

Returns an iterator over module parameters.

This is typically passed to an optimizer.

Parameters **recurse** (*bool*) – if True, then yields parameters of this module and all submodules. Otherwise, yields only parameters that are direct members of this module.

Yields *Parameter* – module parameter

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.

Returns a handle that can be used to remove the added hook by calling `handle.remove()`

Return type `torch.utils.hooks.RemovableHandle`

Warning: The current implementation will not have the presented behavior for complex *Module* that perform many operations. In some failure cases, `grad_input` and `grad_output` will only contain the gradients for a subset of the inputs and outputs. For such *Module*, you should use *torch.Tensor.register_hook()* directly on a specific input or output to get the required gradients.

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, BatchNorms `running_mean` is not a parameter, but is part of the persistent state.

Buffers can be accessed as attributes using given names.

Parameters

- **name** (*string*) – name of the buffer. The buffer can be accessed from this module using the given name
- **tensor** (Tensor) – buffer to be registered.

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 after *forward()* has computed an output. It should have the following signature:

```
hook(module, input, output) -> None
```

The hook should not modify the input or output.

Returns a handle that can be used to remove the added hook by calling `handle.remove()`

Return type `torch.utils.hooks.RemovableHandle`

register_forward_pre_hook(*hook*)
Registers a forward pre-hook on the module.

The hook will be called every time before *forward()* is invoked. It should have the following signature:

```
hook(module, input) -> None
```

The hook should not modify the input.

Returns a handle that can be used to remove the added hook by calling `handle.remove()`

Return type `torch.utils.hooks.RemovableHandle`

register_parameter(*name*, *param*)
Adds a parameter to the module.

The parameter can be accessed as an attribute using given name.

Parameters

- **name** (*string*) – name of the parameter. The parameter can be accessed from this module using the given name
- **param** (Parameter) – parameter to be added to the module.

state_dict(*destination=None*, *prefix=''*, *keep_vars=False*)
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.

Returns a dictionary containing a whole state of the module

Return type dict

Example:

```
>>> module.state_dict().keys()
['bias', 'weight']
```

to(**args*, ***kwargs*)
Moves and/or casts the parameters and buffers.

This can be called as

to(*device=None*, *dtype=None*, *non_blocking=False*)

to(*dtype*, *non_blocking=False*)

to(*tensor*, *non_blocking=False*)

Its signature is similar to *torch.Tensor.to()*, but only accepts floating point desired `dtype` s. In addition, this method will only cast the floating point parameters and buffers to `dtype` (if given). The integral parameters and buffers will be moved `device`, if that is given, but with dtypes unchanged. When `non_blocking` is set, it tries to convert/move asynchronously with respect to the host if possible, e.g., moving CPU Tensors with pinned memory to CUDA devices.

See below for examples.

Note: This method modifies the module in-place.

Parameters

- **device** (`torch.device`) – the desired device of the parameters and buffers in this module
- **dtype** (`torch.dtype`) – the desired floating point type of the floating point parameters and buffers in this module
- **tensor** (`torch.Tensor`) – Tensor whose dtype and device are the desired dtype and device for all parameters and buffers in this module

Returns self

Return type *Module*

Example:

```
>>> linear = nn.Linear(2, 2)
>>> linear.weight
Parameter containing:
tensor([[ 0.1913, -0.3420],
        [-0.5113, -0.2325]])
>>> linear.to(torch.double)
Linear(in_features=2, out_features=2, bias=True)
>>> linear.weight
Parameter containing:
tensor([[ 0.1913, -0.3420],
        [-0.5113, -0.2325]], dtype=torch.float64)
>>> gpu1 = torch.device("cuda:1")
>>> linear.to(gpu1, dtype=torch.half, non_blocking=True)
Linear(in_features=2, out_features=2, bias=True)
>>> linear.weight
Parameter containing:
tensor([[ 0.1914, -0.3420],
        [-0.5112, -0.2324]], dtype=torch.float16, device='cuda:1')
>>> cpu = torch.device("cpu")
>>> linear.to(cpu)
Linear(in_features=2, out_features=2, bias=True)
>>> linear.weight
Parameter containing:
tensor([[ 0.1914, -0.3420],
        [-0.5112, -0.2324]], dtype=torch.float16)
```

train(*mode=True*)

Sets the module in training mode.

This has any effect only on certain modules. See documentations of particular modules for details of their behaviors in training/evaluation mode, if they are affected, e.g. *Dropout*, BatchNorm, etc.

Returns self

Return type *Module*

type(*dst_type*)
Casts all parameters and buffers to dst_type.

Parameters **dst_type** (*type or string*) – the desired type

Returns self

Return type *Module*

zero_grad()
Sets gradients of all model parameters to zero.

16.2.2 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, here 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())
        ]))
```

Shape:

- Input: $(*)$ where * means, any number of additional dimensions
- Output: $(*)$, same shape as the input

16.2.3 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.

Parameters **modules** (*iterable, optional*) – an iterable 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 to the end of the list.

Parameters **module** (nn.Module) – module to append

extend(*modules*)
Appends modules from a Python iterable to the end of the list.

Parameters **modules** (*iterable*) – iterable of modules to append

insert(*index*, *module*)
Insert a given module before a given index in the list.

Parameters

- **index** (*int*) – index to insert.
- **module** (nn.Module) – module to insert

16.2.4 ModuleDict

class torch.nn.**ModuleDict**(*modules=None*)
Holds submodules in a dictionary.

ModuleDict can be indexed like a regular Python dictionary, but modules it contains are properly registered, and will be visible by all *Module* methods.

ModuleDict is an **ordered** dictionary that respects

- the order of insertion, and
- in *update()*, the order of the merged OrderedDict or another *ModuleDict* (the argument to *update()*).

Note that *update()* with other unordered mapping types (e.g., Pythons plain dict) does not preserve the order of the merged mapping.

Parameters **modules** (*iterable, optional*) – a mapping (dictionary) of (string: module) or an iterable of key-value pairs of type (string, module)

Example:

```
class MyModule(nn.Module):
    def __init__(self):
        super(MyModule, self).__init__()
        self.choices = nn.ModuleDict({
                'conv': nn.Conv2d(10, 10, 3),
                'pool': nn.MaxPool2d(3)
        })
```

(continues on next page)

(continued from previous page)

```
        self.activations = nn.ModuleDict([
                ['lrelu', nn.LeakyReLU()],
                ['prelu', nn.PReLU()]
        ])

    def forward(self, x, choice, act):
        x = self.choices[choice](x)
        x = self.activations[act](x)
        return x
```

clear()
Remove all items from the ModuleDict.

items()
Return an iterable of the ModuleDict key/value pairs.

keys()
Return an iterable of the ModuleDict keys.

pop(*key*)
Remove key from the ModuleDict and return its module.

Parameters **key** (*string*) – key to pop from the ModuleDict

update(*modules*)
Update the *ModuleDict* with the key-value pairs from a mapping or an iterable, overwriting existing keys.

Note: If `modules` is an `OrderedDict`, a *ModuleDict*, or an iterable of key-value pairs, the order of new elements in it is preserved.

Parameters **modules** (*iterable*) – a mapping (dictionary) from string to *Module*, or an iterable of key-value pairs of type (string, *Module*)

values()
Return an iterable of the ModuleDict values.

16.2.5 ParameterList

class torch.nn.**ParameterList**(*parameters=None*)
Holds parameters 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.

Parameters **parameters** (*iterable, optional*) – an iterable of *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)])
```

(continues on next page)

(continued from previous page)

```
    def forward(self, x):
        # ParameterList 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.

Parameters **parameter** (nn.Parameter) – parameter to append

extend(*parameters*)
Appends parameters from a Python iterable to the end of the list.

Parameters **parameters** (*iterable*) – iterable of parameters to append

16.2.6 ParameterDict

class torch.nn.**ParameterDict**(*parameters=None*)
Holds parameters in a dictionary.

ParameterDict can be indexed like a regular Python dictionary, but parameters it contains are properly registered, and will be visible by all Module methods.

ParameterDict is an **ordered** dictionary that respects

- the order of insertion, and
- in *update()*, the order of the merged OrderedDict or another *ParameterDict* (the argument to *update()*).

Note that *update()* with other unordered mapping types (e.g., Pythons plain dict) does not preserve the order of the merged mapping.

Parameters **parameters** (*iterable, optional*) – a mapping (dictionary) of (string : *Parameter*) or an iterable of key-value pairs of type (string, *Parameter*)

Example:

```
class MyModule(nn.Module):
    def __init__(self):
        super(MyModule, self).__init__()
        self.params = nn.ParameterDict({
                'left': nn.Parameter(torch.randn(5, 10)),
                'right': nn.Parameter(torch.randn(5, 10))
        })

    def forward(self, x, choice):
        x = self.params[choice].mm(x)
        return x
```

clear()
Remove all items from the ParameterDict.

items()
Return an iterable of the ParameterDict key/value pairs.

keys()
Return an iterable of the ParameterDict keys.

pop(*key*)
Remove key from the ParameterDict and return its parameter.

Parameters **key** (*string*) – key to pop from the ParameterDict

update(*parameters*)
Update the *ParameterDict* with the key-value pairs from a mapping or an iterable, overwriting existing keys.

Note: If `parameters` is an `OrderedDict`, a *ParameterDict*, or an iterable of key-value pairs, the order of new elements in it is preserved.

Parameters **parameters** (*iterable*) – a mapping (dictionary) from string to *Parameter*, or an iterable of key-value pairs of type (string, *Parameter*)

values()
Return an iterable of the ParameterDict values.

16.3 Convolution layers

16.3.1 Conv1d

class torch.nn.**Conv1d**(*in_channels*, *out_channels*, *kernel_size*, *stride=1*, *padding=0*, *dilation=1*, *groups=1*, *bias=True*, *padding_mode='zeros'*)
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 $(N, C_{\text{out}}, L_{\text{out}})$ can be precisely described as:

$$\text{out}(N_i, C_{\text{out}_j}) = \text{bias}(C_{\text{out}_j}) + \sum_{k=0}^{C_{in}-1} \text{weight}(C_{\text{out}_j}, k) \star \text{input}(N_i, k)$$

where $\star$ is the valid cross-correlation operator, N is a batch size, C denotes a number of channels, L is a length of signal sequence.

- `stride` controls the stride for the cross-correlation, a single number or a one-element tuple.
- `padding` controls the amount of implicit zero-paddings on both sides for `padding` number of points.
- `dilation` controls the spacing between the kernel points; also known as the à trous algorithm. It is harder to describe, but this link has a nice visualization of what `dilation` does.
- `groups` controls the connections between inputs and outputs. `in_channels` and `out_channels` must both be divisible by `groups`. For example,
  - 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.
  - At groups= `in_channels`, each input channel is convolved with its own set of filters, of size $\left\lfloor \frac{out\_channels}{in\_channels} \right\rfloor$.

Note: 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.

Note: When *groups == in_channels* and *out_channels == K * in_channels*, where *K* is a positive integer, this operation is also termed in literature as depthwise convolution.

In other words, for an input of size (N, C_{in}, L_{in}), a depthwise convolution with a depthwise multiplier *K*, can be constructed by arguments $(C_{\text{in}} = C_{in}, C_{\text{out}} = C_{in} \times K, ..., \text{groups} = C_{in})$.

Note: In some circumstances when using the CUDA backend with CuDNN, this operator may select a nondeterministic algorithm to increase performance. If this is undesirable, you can try to make the operation deterministic (potentially at a performance cost) by setting `torch.backends.cudnn.deterministic = True`. Please see the notes on *Reproducibility* for background.

Parameters

- **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. Default: 1
- **padding** (*int or tuple, optional*) – Zero-padding added to both sides of the input. Default: 0
- **padding_mode** (*string, optional*) – *zeros*
- **dilation** (*int or tuple, optional*) – Spacing between kernel elements. Default: 1
- **groups** (*int, optional*) – Number of blocked connections from input channels to output channels. Default: 1
- **bias** (*bool, optional*) – If `True`, adds a learnable bias to the output. Default: `True`

Shape:

- Input: (N, C_{in}, L_{in})
- Output: (N, C_{out}, L_{out}) where

$$L_{out} = \left\lfloor \frac{L_{in} + 2 \times \text{padding} - \text{dilation} \times (\text{kernel\_size} - 1) - 1}{\text{stride}} + 1 \right\rfloor$$

Variables

- **weight** (`Tensor`) – the learnable weights of the module of shape $(\text{out\_channels}, \frac{\text{in\_channels}}{\text{groups}}, \text{kernel\_size})$. The values of these weights are sampled from $\mathcal{U}(-\sqrt{k}, \sqrt{k})$ where $k = \frac{1}{C_{\text{in}} * \text{kernel\_size}}$
- **bias** (`Tensor`) – the learnable bias of the module of shape (out_channels). If `bias` is `True`, then the values of these weights are sampled from $\mathcal{U}(-\sqrt{k}, \sqrt{k})$ where $k = \frac{1}{C_{\text{in}} * \text{kernel\_size}}$

Examples:

```
>>> m = nn.Conv1d(16, 33, 3, stride=2)
>>> input = torch.randn(20, 16, 50)
>>> output = m(input)
```

16.3.2 Conv2d

class torch.nn.**Conv2d**(*in_channels*, *out_channels*, *kernel_size*, *stride=1*, *padding=0*, *dilation=1*, *groups=1*, *bias=True*, *padding_mode='zeros'*)

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 $(N, C_{\text{out}}, H_{\text{out}}, W_{\text{out}})$ can be precisely described as:

$$\text{out}(N_i, C_{\text{out}_j}) = \text{bias}(C_{\text{out}_j}) + \sum_{k=0}^{C_{\text{in}}-1} \text{weight}(C_{\text{out}_j}, k) \star \text{input}(N_i, k)$$

where $\star$ is the valid 2D cross-correlation operator, N is a batch size, C denotes a number of channels, H is a height of input planes in pixels, and W is width in pixels.

- `stride` controls the stride for the cross-correlation, a single number or a tuple.
- `padding` controls the amount of implicit zero-paddings on both sides for `padding` number of points for each dimension.
- `dilation` controls the spacing between the kernel points; also known as the à trous algorithm. It is harder to describe, but this link has a nice visualization of what `dilation` does.
- `groups` controls the connections between inputs and outputs. `in_channels` and `out_channels` must both be divisible by `groups`. For example,
  - 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.
  - At groups= `in_channels`, each input channel is convolved with its own set of filters, of size: $\left\lfloor \frac{out\_channels}{in\_channels} \right\rfloor$.

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

---

Note: 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.

Note: When *groups == in_channels* and *out_channels == K * in_channels*, where *K* is a positive integer, this operation is also termed in literature as depthwise convolution.

In other words, for an input of size $(N, C_{in}, H_{in}, W_{in})$, a depthwise convolution with a depthwise multiplier *K*, can be constructed by arguments $(in\_channels = C_{in}, out\_channels = C_{in} \times K, ..., groups = C_{in})$.

Note: In some circumstances when using the CUDA backend with CuDNN, this operator may select a nondeterministic algorithm to increase performance. If this is undesirable, you can try to make the operation deterministic (potentially at a performance cost) by setting `torch.backends.cudnn.deterministic = True`. Please see the notes on *Reproducibility* for background.

Parameters

- **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. Default: 1
- **padding** (*int or tuple, optional*) – Zero-padding added to both sides of the input. Default: 0
- **padding_mode** (*string, optional*) – *zeros*
- **dilation** (*int or tuple, optional*) – Spacing between kernel elements. Default: 1
- **groups** (*int, optional*) – Number of blocked connections from input channels to output channels. Default: 1
- **bias** (*bool, optional*) – If `True`, adds a learnable bias to the output. Default: `True`

Shape:

- Input: $(N, C_{in}, H_{in}, W_{in})$
- Output: $(N, C_{out}, H_{out}, W_{out})$ where

$$H_{out} = \left\lfloor \frac{H_{in} + 2 \times \text{padding}[0] - \text{dilation}[0] \times (\text{kernel\_size}[0] - 1) - 1}{\text{stride}[0]} + 1 \right\rfloor$$

$$W_{out} = \left\lfloor \frac{W_{in} + 2 \times \text{padding}[1] - \text{dilation}[1] \times (\text{kernel\_size}[1] - 1) - 1}{\text{stride}[1]} + 1 \right\rfloor$$

Variables

- **weight** (`Tensor`) – the learnable weights of the module of shape $(\text{out\_channels}, \frac{\text{in\_channels}}{\text{groups}}, \text{kernel\_size}[0], \text{kernel\_size}[1])$. The values of these weights are sampled from $\mathcal{U}(-\sqrt{k}, \sqrt{k})$ where $k = \frac{1}{C_{\text{in}} * \prod_{i=0}^{1} \text{kernel\_size}[i]}$
- **bias** (`Tensor`) – the learnable bias of the module of shape (out_channels). If `bias` is `True`, then the values of these weights are sampled from $\mathcal{U}(-\sqrt{k}, \sqrt{k})$ where $k = \frac{1}{C_{\text{in}} * \prod_{i=0}^{1} \text{kernel\_size}[i]}$

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))
```

(continues on next page)

(continued from previous page)

```
>>> input = torch.randn(20, 16, 50, 100)
>>> output = m(input)
```

16.3.3 Conv3d

class torch.nn.**Conv3d**(*in_channels*, *out_channels*, *kernel_size*, *stride=1*, *padding=0*, *dilation=1*, *groups=1*, *bias=True*, *padding_mode='zeros'*)

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.
- `padding` controls the amount of implicit zero-paddings on both sides for `padding` number of points for each dimension.
- `dilation` controls the spacing between the kernel points; also known as the à trous algorithm. It is harder to describe, but this link has a nice visualization of what `dilation` does.
- `groups` controls the connections between inputs and outputs. `in_channels` and `out_channels` must both be divisible by `groups`. For example,
  - 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.
  - At groups= `in_channels`, each input channel is convolved with its own set of filters, of size $\left\lfloor \frac{out\_channels}{in\_channels} \right\rfloor$.

The parameters `kernel_size`, `stride`, `padding`, `dilation` can either be:

- a single `int` – in which case the same value is used for the depth, 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

---

Note: 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.

Note: When *groups == in_channels* and *out_channels == K * in_channels*, where *K* is a positive integer, this operation is also termed in literature as depthwise convolution.

In other words, for an input of size $(N, C_{in}, D_{in}, H_{in}, W_{in})$, a depthwise convolution with a depthwise multiplier *K*, can be constructed by arguments $(in\_channels = C_{in}, out\_channels = C_{in} \times K, ..., groups = C_{in})$.

Note: In some circumstances when using the CUDA backend with CuDNN, this operator may select a nondeterministic algorithm to increase performance. If this is undesirable, you can try to make the operation de-

terministic (potentially at a performance cost) by setting `torch.backends.cudnn.deterministic = True`. Please see the notes on *Reproducibility* for background.

Parameters

- **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. Default: 1
- **padding** (*int or tuple, optional*) – Zero-padding added to all three sides of the input. Default: 0
- **padding_mode** (*string, optional*) – *zeros*
- **dilation** (*int or tuple, optional*) – Spacing between kernel elements. Default: 1
- **groups** (*int, optional*) – Number of blocked connections from input channels to output channels. Default: 1
- **bias** (*bool, optional*) – If `True`, adds a learnable bias to the output. Default: `True`

Shape:

- Input: $(N, C_{in}, D_{in}, H_{in}, W_{in})$
- Output: $(N, C_{out}, D_{out}, H_{out}, W_{out})$ where

$$D_{out} = \left\lfloor \frac{D_{in} + 2 \times \text{padding}[0] - \text{dilation}[0] \times (\text{kernel\_size}[0] - 1) - 1}{\text{stride}[0]} + 1 \right\rfloor$$

$$H_{out} = \left\lfloor \frac{H_{in} + 2 \times \text{padding}[1] - \text{dilation}[1] \times (\text{kernel\_size}[1] - 1) - 1}{\text{stride}[1]} + 1 \right\rfloor$$

$$W_{out} = \left\lfloor \frac{W_{in} + 2 \times \text{padding}[2] - \text{dilation}[2] \times (\text{kernel\_size}[2] - 1) - 1}{\text{stride}[2]} + 1 \right\rfloor$$

Variables

- **weight** (`Tensor`) – the learnable weights of the module of shape $(\text{out\_channels}, \frac{\text{in\_channels}}{\text{groups}}, \text{kernel\_size}[0], \text{kernel\_size}[1], \text{kernel\_size}[2])$. The values of these weights are sampled from $\mathcal{U}(-\sqrt{k}, \sqrt{k})$ where $k = \frac{1}{C_{\text{in}} * \prod_{i=0}^{2} \text{kernel\_size}[i]}$
- **bias** (`Tensor`) – the learnable bias of the module of shape (out_channels). If `bias` is `True`, then the values of these weights are sampled from $\mathcal{U}(-\sqrt{k}, \sqrt{k})$ where $k = \frac{1}{C_{\text{in}} * \prod_{i=0}^{2} \text{kernel\_size}[i]}$

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 = torch.randn(20, 16, 10, 50, 100)
>>> output = m(input)
```

16.3.4 ConvTranspose1d

class torch.nn.**ConvTranspose1d**(*in_channels*, *out_channels*, *kernel_size*, *stride=1*, *padding=0*, *output_padding=0*, *groups=1*, *bias=True*, *dilation=1*, *padding_mode='zeros'*)

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 also known as a fractionally-strided convolution or a deconvolution (although it is not an actual deconvolution operation).

- `stride` controls the stride for the cross-correlation.
- `padding` controls the amount of implicit zero-paddings on both sides for `dilation * (kernel_size - 1) - padding` number of points. See note below for details.
- `output_padding` controls the additional size added to one side of the output shape. See note below for details.
- `dilation` controls the spacing between the kernel points; also known as the à trous algorithm. It is harder to describe, but this link has a nice visualization of what `dilation` does.
- `groups` controls the connections between inputs and outputs. `in_channels` and `out_channels` must both be divisible by `groups`. For example,
  - 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.
  - At groups= `in_channels`, each input channel is convolved with its own set of filters (of size $\left\lfloor \frac{out\_channels}{in\_channels} \right\rfloor$).

Note: 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.

Note: The `padding` argument effectively adds `dilation * (kernel_size - 1) - padding` amount of zero padding to both sizes of the input. This is set so that when a *`Conv1d`* and a *`ConvTranspose1d`* are initialized with same parameters, they are inverses of each other in regard to the input and output shapes. However, when `stride > 1`, *`Conv1d`* maps multiple input shapes to the same output shape. `output_padding` is provided to resolve this ambiguity by effectively increasing the calculated output shape on one side. Note that `output_padding` is only used to find output shape, but does not actually add zero-padding to output.

Note: In some circumstances when using the CUDA backend with CuDNN, this operator may select a non-deterministic algorithm to increase performance. If this is undesirable, you can try to make the operation deterministic (potentially at a performance cost) by setting `torch.backends.cudnn.deterministic = True`. Please see the notes on *Reproducibility* for background.

Parameters

- **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. Default: 1
- **padding** (*int or tuple, optional*) – `dilation * (kernel_size - 1) - padding` zero-padding will be added to both sides of the input. Default: 0
- **output_padding** (*int or tuple, optional*) – Additional size added to one side of the output shape. Default: 0
- **groups** (*int, optional*) – Number of blocked connections from input channels to output channels. Default: 1
- **bias** (*bool, optional*) – If `True`, adds a learnable bias to the output. Default: `True`
- **dilation** (*int or tuple, optional*) – Spacing between kernel elements. Default: 1

Shape:

- Input: (N, C_{in}, L_{in})
- Output: (N, C_{out}, L_{out}) where

$$L_{out} = (L_{in} - 1) \times \text{stride} - 2 \times \text{padding} + \text{dilation} \times (\text{kernel\_size} - 1) + \text{output\_padding} + 1$$

Variables

- **weight** (`Tensor`) – the learnable weights of the module of shape (in_channels, $\frac{\text{out\_channels}}{\text{groups}}$, kernel_size). The values of these weights are sampled from $\mathcal{U}(-\sqrt{k}, \sqrt{k})$ where $k = \frac{1}{C_{\text{in}} * \text{kernel\_size}}$
- **bias** (`Tensor`) – the learnable bias of the module of shape (out_channels). If `bias` is `True`, then the values of these weights are sampled from $\mathcal{U}(-\sqrt{k}, \sqrt{k})$ where $k = \frac{1}{C_{\text{in}} * \text{kernel\_size}}$

16.3.5 ConvTranspose2d

class torch.nn.**ConvTranspose2d**(*in_channels*, *out_channels*, *kernel_size*, *stride=1*, *padding=0*, *output_padding=0*, *groups=1*, *bias=True*, *dilation=1*, *padding_mode='zeros'*)

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 also known as a fractionally-strided convolution or a deconvolution (although it is not an actual deconvolution operation).

- `stride` controls the stride for the cross-correlation.
- `padding` controls the amount of implicit zero-paddings on both sides for `dilation * (kernel_size - 1) - padding` number of points. See note below for details.
- `output_padding` controls the additional size added to one side of the output shape. See note below for details.
- `dilation` controls the spacing between the kernel points; also known as the à trous algorithm. It is harder to describe, but this link has a nice visualization of what `dilation` does.
- `groups` controls the connections between inputs and outputs. `in_channels` and `out_channels` must both be divisible by `groups`. For example,
  - 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.
- At groups= `in_channels`, each input channel is convolved with its own set of filters (of size $\left\lfloor \frac{out\_channels}{in\_channels} \right\rfloor$).

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 dimensions
- 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

---

Note: 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.

Note: The `padding` argument effectively adds `dilation * (kernel_size - 1) - padding` amount of zero padding to both sizes of the input. This is set so that when a *`Conv2d`* and a *`ConvTranspose2d`* are initialized with same parameters, they are inverses of each other in regard to the input and output shapes. However, when `stride > 1`, *`Conv2d`* maps multiple input shapes to the same output shape. `output_padding` is provided to resolve this ambiguity by effectively increasing the calculated output shape on one side. Note that `output_padding` is only used to find output shape, but does not actually add zero-padding to output.

Note: In some circumstances when using the CUDA backend with CuDNN, this operator may select a nondeterministic algorithm to increase performance. If this is undesirable, you can try to make the operation deterministic (potentially at a performance cost) by setting `torch.backends.cudnn.deterministic = True`. Please see the notes on *Reproducibility* for background.

Parameters

- **`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. Default: 1
- **`padding`** (*`int or tuple, optional`*) – `dilation * (kernel_size - 1) - padding` zero-padding will be added to both sides of each dimension in the input. Default: 0
- **`output_padding`** (*`int or tuple, optional`*) – Additional size added to one side of each dimension in the output shape. Default: 0
- **`groups`** (*`int, optional`*) – Number of blocked connections from input channels to output channels. Default: 1
- **`bias`** (*`bool, optional`*) – If `True`, adds a learnable bias to the output. Default: `True`
- **`dilation`** (*`int or tuple, optional`*) – Spacing between kernel elements. Default: 1

Shape:

- Input: $(N, C_{in}, H_{in}, W_{in})$
- Output: $(N, C_{out}, H_{out}, W_{out})$ where

$$H_{out} = (H_{in} - 1) \times \text{stride}[0] - 2 \times \text{padding}[0] + \text{dilation}[0] \times (\text{kernel\_size}[0] - 1) + \text{output\_padding}[0] + 1$$

$$W_{out} = (W_{in} - 1) \times \text{stride}[1] - 2 \times \text{padding}[1] + \text{dilation}[1] \times (\text{kernel\_size}[1] - 1) + \text{output\_padding}[1] + 1$$

Variables

- **weight** (Tensor) – the learnable weights of the module of shape (in_channels, $\frac{\text{out\_channels}}{\text{groups}}$, kernel_size[0], kernel_size[1]). The values of these weights are sampled from $\mathcal{U}(-\sqrt{k}, \sqrt{k})$ where $k = \frac{1}{C_{\text{in}} * \prod_{i=0}^{1} \text{kernel\_size}[i]}$
- **bias** (Tensor) – the learnable bias of the module of shape (out_channels) If `bias` is `True`, then the values of these weights are sampled from $\mathcal{U}(-\sqrt{k}, \sqrt{k})$ where $k = \frac{1}{C_{\text{in}} * \prod_{i=0}^{1} \text{kernel\_size}[i]}$

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 = torch.randn(20, 16, 50, 100)
>>> output = m(input)
>>> # exact output size can be also specified as an argument
>>> input = 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])
```

16.3.6 ConvTranspose3d

class torch.nn.**ConvTranspose3d**(*in_channels*, *out_channels*, *kernel_size*, *stride=1*, *padding=0*, *output_padding=0*, *groups=1*, *bias=True*, *dilation=1*, *padding_mode='zeros'*)

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 gradient of Conv3d with respect to its input. It is also known as a fractionally-strided convolution or a deconvolution (although it is not an actual deconvolution operation).

- `stride` controls the stride for the cross-correlation.
- `padding` controls the amount of implicit zero-paddings on both sides for `dilation * (kernel_size - 1) - padding` number of points. See note below for details.
- `output_padding` controls the additional size added to one side of the output shape. See note below for details.

- `dilation` controls the spacing between the kernel points; also known as the à trous algorithm. It is harder to describe, but this link has a nice visualization of what `dilation` does.
- `groups` controls the connections between inputs and outputs. `in_channels` and `out_channels` must both be divisible by `groups`. For example,
  - 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.
  - At groups= `in_channels`, each input channel is convolved with its own set of filters (of size $\left\lfloor \frac{out\_channels}{in\_channels} \right\rfloor$).

The parameters `kernel_size`, `stride`, `padding`, `output_padding` can either be:

- a single `int` – in which case the same value is used for the depth, height and width dimensions
- 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

---

Note: 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.

Note: The `padding` argument effectively adds `dilation * (kernel_size - 1) - padding` amount of zero padding to both sizes of the input. This is set so that when a *`Conv3d`* and a *`ConvTranspose3d`* are initialized with same parameters, they are inverses of each other in regard to the input and output shapes. However, when `stride > 1`, *`Conv3d`* maps multiple input shapes to the same output shape. `output_padding` is provided to resolve this ambiguity by effectively increasing the calculated output shape on one side. Note that `output_padding` is only used to find output shape, but does not actually add zero-padding to output.

Note: In some circumstances when using the CUDA backend with CuDNN, this operator may select a nondeterministic algorithm to increase performance. If this is undesirable, you can try to make the operation deterministic (potentially at a performance cost) by setting `torch.backends.cudnn.deterministic = True`. Please see the notes on *Reproducibility* for background.

Parameters

- **`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. Default: 1
- **`padding`** (*`int or tuple, optional`*) – `dilation * (kernel_size - 1) - padding` zero-padding will be added to both sides of each dimension in the input. Default: 0
- **`output_padding`** (*`int or tuple, optional`*) – Additional size added to one side of each dimension in the output shape. Default: 0
- **`groups`** (*`int, optional`*) – Number of blocked connections from input channels to output channels. Default: 1

- **bias** (*bool, optional*) – If `True`, adds a learnable bias to the output. Default: `True`
- **dilation** (*int or tuple, optional*) – Spacing between kernel elements. Default: 1

Shape:

- Input: $(N, C_{in}, D_{in}, H_{in}, W_{in})$
- Output: $(N, C_{out}, D_{out}, H_{out}, W_{out})$ where

$$D_{out} = (D_{in} - 1) \times \text{stride}[0] - 2 \times \text{padding}[0] + \text{dilation}[0] \times (\text{kernel\_size}[0] - 1) + \text{output\_padding}[0] + 1$$

$$H_{out} = (H_{in} - 1) \times \text{stride}[1] - 2 \times \text{padding}[1] + \text{dilation}[1] \times (\text{kernel\_size}[1] - 1) + \text{output\_padding}[1] + 1$$

$$W_{out} = (W_{in} - 1) \times \text{stride}[2] - 2 \times \text{padding}[2] + \text{dilation}[2] \times (\text{kernel\_size}[2] - 1) + \text{output\_padding}[2] + 1$$

Variables

- **weight** (`Tensor`) – the learnable weights of the module of shape (in_channels, $\frac{\text{out\_channels}}{\text{groups}}$, kernel_size[0], kernel_size[1], kernel_size[2]). The values of these weights are sampled from $\mathcal{U}(-\sqrt{k}, \sqrt{k})$ where $k = \frac{1}{C_{\text{in}} * \prod_{i=0}^{2} \text{kernel\_size}[i]}$
- **bias** (`Tensor`) – the learnable bias of the module of shape (out_channels) If `bias` is `True`, then the values of these weights are sampled from $\mathcal{U}(-\sqrt{k}, \sqrt{k})$ where $k = \frac{1}{C_{\text{in}} * \prod_{i=0}^{2} \text{kernel\_size}[i]}$

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.ConvTranspose3d(16, 33, (3, 5, 2), stride=(2, 1, 1), padding=(0, 4, 2))
>>> input = torch.randn(20, 16, 10, 50, 100)
>>> output = m(input)
```

16.3.7 Unfold

class `torch.nn.`**Unfold**(*kernel_size*, *dilation=1*, *padding=0*, *stride=1*)

Extracts sliding local blocks from a batched input tensor.

Consider an batched `input` tensor of shape $(N, C, *)$, where N is the batch dimension, C is the channel dimension, and $*$ represent arbitrary spatial dimensions. This operation flattens each sliding `kernel_size`-sized block within the spatial dimensions of `input` into a column (i.e., last dimension) of a 3-D `output` tensor of shape $(N, C \times \prod(\text{kernel\_size}), L)$, where $C \times \prod(\text{kernel\_size})$ is the total number of values within each block (a block has $\prod(\text{kernel\_size})$ spatial locations each containing a C-channeled vector), and L is the total number of such blocks:

$$L = \prod_{d} \left\lfloor \frac{\text{spatial\_size}[d] + 2 \times \text{padding}[d] - \text{dilation}[d] \times (\text{kernel\_size}[d] - 1) - 1}{\text{stride}[d]} + 1 \right\rfloor,$$

where spatial_size is formed by the spatial dimensions of `input` ($*$ above), and d is over all spatial dimensions.

Therefore, indexing `output` at the last dimension (column dimension) gives all values within a certain block.

The `padding`, `stride` and `dilation` arguments specify how the sliding blocks are retrieved.

- `stride` controls the stride for the sliding blocks.

- `padding` controls the amount of implicit zero-paddings on both sides for `padding` number of points for each dimension before reshaping.
- `dilation` controls the spacing between the kernel points; also known as the à trous algorithm. It is harder to describe, but this link has a nice visualization of what `dilation` does.

Parameters

- **kernel_size** (*int or tuple*) – the size of the sliding blocks
- **stride** (*int or tuple, optional*) – the stride of the sliding blocks in the input spatial dimensions. Default: 1
- **padding** (*int or tuple, optional*) – implicit zero padding to be added on both sides of input. Default: 0
- **dilation** (*int or tuple, optional*) – a parameter that controls the stride of elements within the neighborhood. Default: 1

- If `kernel_size`, `dilation`, `padding` or `stride` is an int or a tuple of length 1, their values will be replicated across all spatial dimensions.
- For the case of two input spatial dimensions this operation is sometimes called `im2col`.

Note: *Fold* calculates each combined value in the resulting large tensor by summing all values from all containing blocks. *Unfold* extracts the values in the local blocks by copying from the large tensor. So, if the blocks overlap, they are not inverses of each other.

Warning: Currently, only 4-D input tensors (batched image-like tensors) are supported.

Shape:

- Input: $(N, C, *)$
- Output: $(N, C \times \prod(\text{kernel\_size}), L)$ as described above

Examples:

```
>>> unfold = nn.Unfold(kernel_size=(2, 3))
>>> input = torch.randn(2, 5, 3, 4)
>>> output = unfold(input)
>>> # each patch contains 30 values (2x3=6 vectors, each of 5 channels)
>>> # 4 blocks (2x3 kernels) in total in the 3x4 input
>>> output.size()
torch.Size([2, 30, 4])

>>> # Convolution is equivalent with Unfold + Matrix Multiplication + Fold (or␣
↪view to output shape)
>>> inp = torch.randn(1, 3, 10, 12)
>>> w = torch.randn(2, 3, 4, 5)
>>> inp_unf = torch.nn.functional.unfold(inp, (4, 5))
>>> out_unf = inp_unf.transpose(1, 2).matmul(w.view(w.size(0), -1).t()).
↪transpose(1, 2)
>>> out = torch.nn.functional.fold(out_unf, (7, 8), (1, 1))
>>> # or equivalently (and avoiding a copy),
```

(continues on next page)

(continued from previous page)

```
>>> # out = out_unf.view(1, 2, 7, 8)
>>> (torch.nn.functional.conv2d(inp, w) - out).abs().max()
tensor(1.9073e-06)
```

16.3.8 Fold

class torch.nn.**Fold**(*output_size*, *kernel_size*, *dilation=1*, *padding=0*, *stride=1*)

Combines an array of sliding local blocks into a large containing tensor.

Consider a batched `input` tensor containing sliding local blocks, e.g., patches of images, of shape $(N, C \times \prod(\text{kernel\_size}), L)$, where N is batch dimension, $C \times \prod(\text{kernel\_size})$ is the number of values within a block (a block has $\prod(\text{kernel\_size})$ spatial locations each containing a C-channeled vector), and L is the total number of blocks. (This is exactly the same specification as the output shape of *Unfold*.) This operation combines these local blocks into the large `output` tensor of shape $(N, C, \text{output\_size}[0], \text{output\_size}[1], \dots)$ by summing the overlapping values. Similar to *Unfold*, the arguments must satisfy

$$L = \prod_d \left\lfloor \frac{\text{output\_size}[d] + 2 \times \text{padding}[d] - \text{dilation}[d] \times (\text{kernel\_size}[d] - 1) - 1}{\text{stride}[d]} + 1 \right\rfloor,$$

where d is over all spatial dimensions.

- `output_size` describes the spatial shape of the large containing tensor of the sliding local blocks. It is useful to resolve the ambiguity when multiple input shapes map to same number of sliding blocks, e.g., with `stride > 0`.

The `padding`, `stride` and `dilation` arguments specify how the sliding blocks are retrieved.

- `stride` controls the stride for the sliding blocks.
- `padding` controls the amount of implicit zero-paddings on both sides for `padding` number of points for each dimension before reshaping.
- `dilation` controls the spacing between the kernel points; also known as the à trous algorithm. It is harder to describe, but this link has a nice visualization of what `dilation` does.

Parameters

- **output_size** (*int or tuple*) – the shape of the spatial dimensions of the output (i.e., `output.sizes()[2:]`)
- **kernel_size** (*int or tuple*) – the size of the sliding blocks
- **stride** (*int or tuple*) – the stride of the sliding blocks in the input spatial dimensions. Default: 1
- **padding** (*int or tuple, optional*) – implicit zero padding to be added on both sides of input. Default: 0
- **dilation** (*int or tuple, optional*) – a parameter that controls the stride of elements within the neighborhood. Default: 1

- If `output_size`, `kernel_size`, `dilation`, `padding` or `stride` is an int or a tuple of length 1 then their values will be replicated across all spatial dimensions.
- For the case of two output spatial dimensions this operation is sometimes called `col2im`.

Note: *Fold* calculates each combined value in the resulting large tensor by summing all values from all containing blocks. *Unfold* extracts the values in the local blocks by copying from the large tensor. So, if the blocks overlap, they are not inverses of each other.

Warning: Currently, only 4-D output tensors (batched image-like tensors) are supported.

Shape:

- Input: $(N, C \times \prod(\text{kernel\_size}), L)$
- Output: $(N, C, \text{output\_size}[0], \text{output\_size}[1], \dots)$ as described above

Examples:

```
>>> fold = nn.Fold(output_size=(4, 5), kernel_size=(2, 2))
>>> input = torch.randn(1, 3 * 2 * 2, 12)
>>> output = fold(input)
>>> output.size()
torch.Size([1, 3, 4, 5])
```

16.4 Pooling layers

16.4.1 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:

$$out(N_i, C_j, k) = \max_{m=0,\dots,\text{kernel\_size}-1} input(N_i, C_j, stride \times k + m)$$

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.

Parameters

- **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 for *torch.nn.MaxUnpool1d* 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

$$L_{out} = \left\lfloor \frac{L_{in} + 2 \times \text{padding} - \text{dilation} \times (\text{kernel\_size} - 1) - 1}{\text{stride}} + 1 \right\rfloor$$

Examples:

```
>>> # pool of size=3, stride=2
>>> m = nn.MaxPool1d(3, stride=2)
>>> input = torch.randn(20, 16, 50)
>>> output = m(input)
```

16.4.2 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:

$$out(N_i, C_j, h, w) = \max_{m=0,\ldots,kH-1} \max_{n=0,\ldots,kW-1} \text{input}(N_i, C_j, \text{stride}[0] \times h + m, \text{stride}[1] \times w + n)$$

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

Parameters

- **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 for *torch.nn.MaxUnpool2d* 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

$$H_{out} = \left\lfloor \frac{H_{in} + 2 * \text{padding}[0] - \text{dilation}[0] \times (\text{kernel\_size}[0] - 1) - 1}{\text{stride}[0]} + 1 \right\rfloor$$

$$W_{out} = \left\lfloor \frac{W_{in} + 2 * \text{padding}[1] - \text{dilation}[1] \times (\text{kernel\_size}[1] - 1) - 1}{\text{stride}[1]} + 1 \right\rfloor$$

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 = torch.randn(20, 16, 50, 32)
>>> output = m(input)
```

16.4.3 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:

$$\text{out}(N_i, C_j, d, h, w) = \max_{k=0,\ldots,kD-1} \max_{m=0,\ldots,kH-1} \max_{n=0,\ldots,kW-1} \text{input}(N_i, C_j, \text{stride}[0] \times d + k, \text{stride}[1] \times h + m, \text{stride}[2] \times w + n)$$

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 depth, 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

Parameters

- **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 all three 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 for *torch.nn.MaxUnpool3d* 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

$$D_{out} = \left\lfloor \frac{D_{in} + 2 \times \text{padding}[0] - \text{dilation}[0] \times (\text{kernel\_size}[0] - 1) - 1}{\text{stride}[0]} + 1 \right\rfloor$$

$$H_{out} = \left\lfloor \frac{H_{in} + 2 \times \text{padding}[1] - \text{dilation}[1] \times (\text{kernel\_size}[1] - 1) - 1}{\text{stride}[1]} + 1 \right\rfloor$$

$$W_{out} = \left\lfloor \frac{W_{in} + 2 \times \text{padding}[2] - \text{dilation}[2] \times (\text{kernel\_size}[2] - 1) - 1}{\text{stride}[2]} + 1 \right\rfloor$$

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 = torch.randn(20, 16, 50,44, 31)
>>> output = m(input)
```

16.4.4 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.

Note: *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.

Parameters

- **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): the targeted output size

Shape:

- Input: (N, C, H_{in})
- Output: (N, C, H_{out}), where

$$H_{out} = (H_{in} - 1) \times \text{stride}[0] - 2 \times \text{padding}[0] + \text{kernel\_size}[0]$$

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 = torch.tensor([[[1., 2, 3, 4, 5, 6, 7, 8]]])
>>> output, indices = pool(input)
>>> unpool(output, indices)
```

(continues on next page)

(continued from previous page)

```
tensor([[[ 0.,  2.,  0.,  4.,  0.,  6.,  0., 8.]]])

>>> # Example showcasing the use of output_size
>>> input = torch.tensor([[[1., 2, 3, 4, 5, 6, 7, 8, 9]]])
>>> output, indices = pool(input)
>>> unpool(output, indices, output_size=input.size())
tensor([[[ 0.,  2.,  0.,  4.,  0.,  6.,  0., 8.,  0.]]])

>>> unpool(output, indices)
tensor([[[ 0.,  2.,  0.,  4.,  0.,  6.,  0., 8.]]])
```

16.4.5 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.

Note: *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.

Parameters

- **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): the targeted output size

Shape:

- Input: (N, C, H_{in}, W_{in})
- Output: (N, C, H_{out}, W_{out}), where

$$H_{out} = (H_{in} - 1) \times \text{stride}[0] - 2 \times \text{padding}[0] + \text{kernel\_size}[0]$$

$$W_{out} = (W_{in} - 1) \times \text{stride}[1] - 2 \times \text{padding}[1] + \text{kernel\_size}[1]$$

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 = 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)
tensor([[[[  0.,   0.,   0.,   0.],
          [  0.,   6.,   0.,   8.],
          [  0.,   0.,   0.,   0.],
          [  0.,  14.,   0.,  16.]]]])

>>> # specify a different output size than input size
>>> unpool(output, indices, output_size=torch.Size([1, 1, 5, 5]))
tensor([[[[  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.]]]])
```

16.4.6 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.

Note: *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.

Parameters

- **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): the targeted output size

Shape:

- Input: $(N, C, D_{in}, H_{in}, W_{in})$

- Output: $(N, C, D_{out}, H_{out}, W_{out})$, where

$$D_{out} = (D_{in} - 1) \times \text{stride}[0] - 2 \times \text{padding}[0] + \text{kernel\_size}[0]$$

$$H_{out} = (H_{in} - 1) \times \text{stride}[1] - 2 \times \text{padding}[1] + \text{kernel\_size}[1]$$

$$W_{out} = (W_{in} - 1) \times \text{stride}[2] - 2 \times \text{padding}[2] + \text{kernel\_size}[2]$$

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(torch.randn(20, 16, 51, 33, 15))
>>> unpooled_output = unpool(output, indices)
>>> unpooled_output.size()
torch.Size([20, 16, 51, 33, 15])
```

16.4.7 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:

$$\text{out}(N_i, C_j, l) = \frac{1}{k} \sum_{m=0}^{k-1} \text{input}(N_i, C_j, \text{stride} \times l + m)$$

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.

Parameters

- **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

$$L_{out} = \left\lfloor \frac{L_{in} + 2 \times \text{padding} - \text{kernel\_size}}{\text{stride}} + 1 \right\rfloor$$

Examples:

```
>>> # pool with window of size=3, stride=2
>>> m = nn.AvgPool1d(3, stride=2)
>>> m(torch.tensor([[[1.,2,3,4,5,6,7]]]))
tensor([[[ 2.,  4.,  6.]]])
```

16.4.8 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:

$$out(N_i, C_j, h, w) = \frac{1}{kH * kW} \sum_{m=0}^{kH-1} \sum_{n=0}^{kW-1} input(N_i, C_j, stride[0] \times h + m, stride[1] \times w + n)$$

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

Parameters

- **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

$$H_{out} = \left\lfloor \frac{H_{in} + 2 \times \text{padding}[0] - \text{kernel\_size}[0]}{\text{stride}[0]} + 1 \right\rfloor$$

$$W_{out} = \left\lfloor \frac{W_{in} + 2 \times \text{padding}[1] - \text{kernel\_size}[1]}{\text{stride}[1]} + 1 \right\rfloor$$

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 = torch.randn(20, 16, 50, 32)
>>> output = m(input)
```

16.4.9 AvgPool3d

class torch.nn.**AvgPool3d**(*kernel_size*, *stride=None*, *padding=0*, *ceil_mode=False*, *count_include_pad=True*)

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:

$$\text{out}(N_i, C_j, d, h, w) = \sum_{k=0}^{kD-1} \sum_{m=0}^{kH-1} \sum_{n=0}^{kW-1} \frac{\text{input}(N_i, C_j, \text{stride}[0] \times d + k, \text{stride}[1] \times h + m, \text{stride}[2] \times w + n)}{kD \times kH \times kW}$$

If `padding` is non-zero, then the input is implicitly zero-padded on all three sides for `padding` number of points.

The parameters `kernel_size`, `stride` can either be:

- a single `int` – in which case the same value is used for the depth, 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

Parameters

- **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 all three 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: $(N, C, D_{in}, H_{in}, W_{in})$
- Output: $(N, C, D_{out}, H_{out}, W_{out})$, where

$$D_{out} = \left\lfloor \frac{D_{in} + 2 \times \text{padding}[0] - \text{kernel\_size}[0]}{\text{stride}[0]} + 1 \right\rfloor$$

$$H_{out} = \left\lfloor \frac{H_{in} + 2 \times \text{padding}[1] - \text{kernel\_size}[1]}{\text{stride}[1]} + 1 \right\rfloor$$

$$W_{out} = \left\lfloor \frac{W_{in} + 2 \times \text{padding}[2] - \text{kernel\_size}[2]}{\text{stride}[2]} + 1 \right\rfloor$$

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 = torch.randn(20, 16, 50,44, 31)
>>> output = m(input)
```

16.4.10 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.

Fractional MaxPooling is described in detail in the paper Fractional MaxPooling by Ben Graham

The max-pooling operation is applied in $kH \times kW$ 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.

Parameters

- **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, 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 = torch.randn(20, 16, 50, 32)
>>> output = m(input)
```

16.4.11 LPPool1d

class torch.nn.**LPPool1d**(*norm_type*, *kernel_size*, *stride=None*, *ceil_mode=False*)

Applies a 1D power-average pooling over an input signal composed of several input planes.

On each window, the function computed is:

$$f(X) = \sqrt[p]{\sum_{x \in X} x^p}$$

- At p = ∞, one gets Max Pooling
- At p = 1, one gets Sum Pooling (which is proportional to Average Pooling)

Note: If the sum to the power of *p* is zero, the gradient of this function is not defined. This implementation will set the gradient to zero in this case.

Parameters

- **kernel_size** – a single int, the size of the window
- **stride** – a single int, 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, L_{in})
- Output: (N, C, L_{out}), where

$$L_{out} = \left\lfloor \frac{L_{in} + 2 \times \text{padding} - \text{kernel\_size}}{\text{stride}} + 1 \right\rfloor$$

Examples::

```
>>> # power-2 pool of window of length 3, with stride 2.
>>> m = nn.LPPool1d(2, 3, stride=2)
>>> input = torch.randn(20, 16, 50)
>>> output = m(input)
```

16.4.12 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:

$$f(X) = \sqrt[p]{\sum_{x \in X} x^p}$$

- At p = ∞, one gets Max Pooling
- At p = 1, one gets Sum Pooling (which is proportional to 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

---

Note: If the sum to the power of *p* is zero, the gradient of this function is not defined. This implementation will set the gradient to zero in this case.

Parameters

- **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

$$H_{out} = \left\lfloor \frac{H_{in} + 2 \times \text{padding}[0] - \text{dilation}[0] \times (\text{kernel\_size}[0] - 1) - 1}{\text{stride}[0]} + 1 \right\rfloor$$

$$W_{out} = \left\lfloor \frac{W_{in} + 2 \times \text{padding}[1] - \text{dilation}[1] \times (\text{kernel\_size}[1] - 1) - 1}{\text{stride}[1]} + 1 \right\rfloor$$

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 = torch.randn(20, 16, 50, 32)
>>> output = m(input)
```

16.4.13 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.

Parameters

- **output_size** – the target output size H
- **return_indices** – if `True`, will return the indices along with the outputs. Useful to pass to nn.MaxUnpool1d. Default: `False`

Examples

```
>>> # target output size of 5
>>> m = nn.AdaptiveMaxPool1d(5)
>>> input = torch.randn(1, 64, 8)
>>> output = m(input)
```

16.4.14 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.

Parameters

- **output_size** – the target output size of the image of the form H x W. Can be a tuple (H, W) or a single H for a square image H x H. H and W can be either a `int`, or `None` which means the size will be the same as that of the input.
- **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 = torch.randn(1, 64, 8, 9)
>>> output = m(input)
>>> # target output size of 7x7 (square)
>>> m = nn.AdaptiveMaxPool2d(7)
>>> input = torch.randn(1, 64, 10, 9)
>>> output = m(input)
>>> # target output size of 10x7
>>> m = nn.AdaptiveMaxPool2d((None, 7))
>>> input = torch.randn(1, 64, 10, 9)
>>> output = m(input)
```

16.4.15 AdaptiveMaxPool3d

class torch.nn.**AdaptiveMaxPool3d**(*output_size*, *return_indices=False*)

Applies a 3D adaptive max pooling over an input signal composed of several input planes.

The output is of size D x H x W, for any input size. The number of output features is equal to the number of input planes.

Parameters

- **output_size** – the target output size of the image of the form D x H x W. Can be a tuple (D, H, W) or a single D for a cube D x D x D. D, H and W can be either a `int`, or `None` which means the size will be the same as that of the input.
- **return_indices** – if `True`, will return the indices along with the outputs. Useful to pass to nn.MaxUnpool3d. Default: `False`

Examples

```
>>> # target output size of 5x7x9
>>> m = nn.AdaptiveMaxPool3d((5,7,9))
>>> input = torch.randn(1, 64, 8, 9, 10)
>>> output = m(input)
>>> # target output size of 7x7x7 (cube)
>>> m = nn.AdaptiveMaxPool3d(7)
>>> input = torch.randn(1, 64, 10, 9, 8)
>>> output = m(input)
>>> # target output size of 7x9x8
>>> m = nn.AdaptiveMaxPool3d((7, None, None))
>>> input = torch.randn(1, 64, 10, 9, 8)
>>> output = m(input)
```

16.4.16 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.

Parameters **output_size** – the target output size H

Examples

```
>>> # target output size of 5
>>> m = nn.AdaptiveAvgPool1d(5)
>>> input = torch.randn(1, 64, 8)
>>> output = m(input)
```

16.4.17 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.

Parameters **output_size** – the target output size of the image of the form H x W. Can be a tuple (H, W) or a single H for a square image H x H. H and W can be either a `int`, or `None` which means the size will be the same as that of the input.

Examples

```
>>> # target output size of 5x7
>>> m = nn.AdaptiveAvgPool2d((5,7))
>>> input = torch.randn(1, 64, 8, 9)
>>> output = m(input)
>>> # target output size of 7x7 (square)
>>> m = nn.AdaptiveAvgPool2d(7)
>>> input = torch.randn(1, 64, 10, 9)
>>> output = m(input)
>>> # target output size of 10x7
>>> m = nn.AdaptiveMaxPool2d((None, 7))
>>> input = torch.randn(1, 64, 10, 9)
>>> output = m(input)
```

16.4.18 AdaptiveAvgPool3d

class torch.nn.**AdaptiveAvgPool3d**(*output_size*)

Applies a 3D adaptive average pooling over an input signal composed of several input planes.

The output is of size D x H x W, for any input size. The number of output features is equal to the number of input planes.

Parameters **output_size** – the target output size of the form D x H x W. Can be a tuple (D, H, W) or a single number D for a cube D x D x D. D, H and W can be either a `int`, or `None` which means the size will be the same as that of the input.

Examples

```
>>> # target output size of 5x7x9
>>> m = nn.AdaptiveAvgPool3d((5,7,9))
>>> input = torch.randn(1, 64, 8, 9, 10)
```

(continues on next page)

(continued from previous page)

```
>>> output = m(input)
>>> # target output size of 7x7x7 (cube)
>>> m = nn.AdaptiveAvgPool3d(7)
>>> input = torch.randn(1, 64, 10, 9, 8)
>>> output = m(input)
>>> # target output size of 7x9x8
>>> m = nn.AdaptiveMaxPool3d((7, None, None))
>>> input = torch.randn(1, 64, 10, 9, 8)
>>> output = m(input)
```

16.5 Padding layers

16.5.1 ReflectionPad1d

class torch.nn.**ReflectionPad1d**(*padding*)

Pads the input tensor using the reflection of the input boundary.

For *N*-dimensional padding, use *torch.nn.functional.pad()*.

Parameters **padding** (*int, tuple*) – the size of the padding. If is *int*, uses the same padding in all boundaries. If a 2-*tuple*, uses (padding_left, padding_right)

Shape:

- Input: (N, C, W_{in})
- Output: (N, C, W_{out}) where

  $W_{out} = W_{in} + \text{padding\_left} + \text{padding\_right}$

Examples:

```
>>> m = nn.ReflectionPad1d(2)
>>> input = torch.arange(8, dtype=torch.float).reshape(1, 2, 4)
>>> input
tensor([[[0., 1., 2., 3.],
         [4., 5., 6., 7.]]])
>>> m(input)
tensor([[[2., 1., 0., 1., 2., 3., 2., 1.],
         [6., 5., 4., 5., 6., 7., 6., 5.]]])
>>> # using different paddings for different sides
>>> m = nn.ReflectionPad1d((3, 1))
>>> m(input)
tensor([[[3., 2., 1., 0., 1., 2., 3., 2.],
         [7., 6., 5., 4., 5., 6., 7., 6.]]])
```

16.5.2 ReflectionPad2d

class torch.nn.**ReflectionPad2d**(*padding*)

Pads the input tensor using the reflection of the input boundary.

For *N*-dimensional padding, use *torch.nn.functional.pad()*.

Parameters **padding** (*int, tuple*) – the size of the padding. If is *int*, uses the same padding in all boundaries. If a 4-*tuple*, uses (padding_left, padding_right, padding_top, padding_bottom)

Shape:

- Input: (N, C, H_{in}, W_{in})
- Output: (N, C, H_{out}, W_{out}) where

  $H_{out} = H_{in} + \text{padding\_top} + \text{padding\_bottom}$

  $W_{out} = W_{in} + \text{padding\_left} + \text{padding\_right}$

Examples:

```
>>> m = nn.ReflectionPad2d(2)
>>> input = torch.arange(9, dtype=torch.float).reshape(1, 1, 3, 3)
>>> input
tensor([[[[0., 1., 2.],
          [3., 4., 5.],
          [6., 7., 8.]]]])
>>> m(input)
tensor([[[[8., 7., 6., 7., 8., 7., 6.],
          [5., 4., 3., 4., 5., 4., 3.],
          [2., 1., 0., 1., 2., 1., 0.],
          [5., 4., 3., 4., 5., 4., 3.],
          [8., 7., 6., 7., 8., 7., 6.],
          [5., 4., 3., 4., 5., 4., 3.],
          [2., 1., 0., 1., 2., 1., 0.]]]])
>>> # using different paddings for different sides
>>> m = nn.ReflectionPad2d((1, 1, 2, 0))
>>> m(input)
tensor([[[[7., 6., 7., 8., 7.],
          [4., 3., 4., 5., 4.],
          [1., 0., 1., 2., 1.],
          [4., 3., 4., 5., 4.],
          [7., 6., 7., 8., 7.]]]])
```

16.5.3 ReplicationPad1d

class torch.nn.**ReplicationPad1d**(*padding*)

Pads the input tensor using replication of the input boundary.

For *N*-dimensional padding, use *torch.nn.functional.pad()*.

Parameters **padding** (*int, tuple*) – the size of the padding. If is *int*, uses the same padding in all boundaries. If a 2-*tuple*, uses (padding_left, padding_right)

Shape:

- Input: (N, C, W_{in})
- Output: (N, C, W_{out}) where

  $W_{out} = W_{in} + \text{padding\_left} + \text{padding\_right}$

Examples:

```
>>> m = nn.ReplicationPad1d(2)
>>> input = torch.arange(8, dtype=torch.float).reshape(1, 2, 4)
>>> input
tensor([[[0., 1., 2., 3.],
         [4., 5., 6., 7.]]])
>>> m(input)
tensor([[[0., 0., 0., 1., 2., 3., 3., 3.],
         [4., 4., 4., 5., 6., 7., 7., 7.]]])
>>> # using different paddings for different sides
>>> m = nn.ReplicationPad1d((3, 1))
>>> m(input)
tensor([[[0., 0., 0., 0., 1., 2., 3., 3.],
         [4., 4., 4., 4., 5., 6., 7., 7.]]])
```

16.5.4 ReplicationPad2d

class torch.nn.**ReplicationPad2d**(*padding*)

Pads the input tensor using replication of the input boundary.

For *N*-dimensional padding, use *torch.nn.functional.pad()*.

Parameters **padding** (*int, tuple*) – the size of the padding. If is *int*, uses the same padding in all boundaries. If a 4-*tuple*, uses (padding_left, padding_right, padding_top, padding_bottom)

Shape:

- Input: (N, C, H_{in}, W_{in})
- Output: (N, C, H_{out}, W_{out}) where

  $H_{out} = H_{in} + \text{padding\_top} + \text{padding\_bottom}$

  $W_{out} = W_{in} + \text{padding\_left} + \text{padding\_right}$

Examples:

```
>>> m = nn.ReplicationPad2d(2)
>>> input = torch.arange(9, dtype=torch.float).reshape(1, 1, 3, 3)
>>> input
tensor([[[[0., 1., 2.],
          [3., 4., 5.],
          [6., 7., 8.]]]])
>>> m(input)
tensor([[[[0., 0., 0., 1., 2., 2., 2.],
          [0., 0., 0., 1., 2., 2., 2.],
          [0., 0., 0., 1., 2., 2., 2.],
          [3., 3., 3., 4., 5., 5., 5.],
          [6., 6., 6., 7., 8., 8., 8.],
          [6., 6., 6., 7., 8., 8., 8.],
          [6., 6., 6., 7., 8., 8., 8.]]]])
>>> # using different paddings for different sides
>>> m = nn.ReplicationPad2d((1, 1, 2, 0))
>>> m(input)
tensor([[[[0., 0., 1., 2., 2.],
          [0., 0., 1., 2., 2.],
          [0., 0., 1., 2., 2.],
          [3., 3., 4., 5., 5.],
          [6., 6., 7., 8., 8.]]]])
```

16.5.5 ReplicationPad3d

class torch.nn.**ReplicationPad3d**(*padding*)

Pads the input tensor using replication of the input boundary.

For *N*-dimensional padding, use *torch.nn.functional.pad()*.

Parameters **padding** (*int, tuple*) – the size of the padding. If is *int*, uses the same padding in all boundaries. If a 6-*tuple*, uses (padding_left, padding_right, padding_top, padding_bottom, padding_front, padding_back)

Shape:

- Input: $(N, C, D_{in}, H_{in}, W_{in})$
- Output: $(N, C, D_{out}, H_{out}, W_{out})$ where

  $D_{out} = D_{in} + \text{padding\_front} + \text{padding\_back}$

  $H_{out} = H_{in} + \text{padding\_top} + \text{padding\_bottom}$

  $W_{out} = W_{in} + \text{padding\_left} + \text{padding\_right}$

Examples:

```
>>> m = nn.ReplicationPad3d(3)
>>> input = torch.randn(16, 3, 8, 320, 480)
>>> output = m(input)
>>> # using different paddings for different sides
>>> m = nn.ReplicationPad3d((3, 3, 6, 6, 1, 1))
>>> output = m(input)
```

16.5.6 ZeroPad2d

class torch.nn.**ZeroPad2d**(*padding*)

Pads the input tensor boundaries with zero.

For *N*-dimensional padding, use *torch.nn.functional.pad()*.

Parameters **padding** (*int, tuple*) – the size of the padding. If is *int*, uses the same padding in all boundaries. If a 4-*tuple*, uses (padding_left, padding_right, padding_top, padding_bottom)

Shape:

- Input: (N, C, H_{in}, W_{in})
- Output: (N, C, H_{out}, W_{out}) where

  $H_{out} = H_{in} + \text{padding\_top} + \text{padding\_bottom}$

  $W_{out} = W_{in} + \text{padding\_left} + \text{padding\_right}$

Examples:

```
>>> m = nn.ZeroPad2d(2)
>>> input = torch.randn(1, 1, 3, 3)
>>> input
tensor([[[[-0.1678, -0.4418,  1.9466],
          [ 0.9604, -0.4219, -0.5241],
          [-0.9162, -0.5436, -0.6446]]]])
```

(continues on next page)

(continued from previous page)

```
>>> m(input)
tensor([[[[ 0.0000,  0.0000,  0.0000,  0.0000,  0.0000,  0.0000,  0.0000],
          [ 0.0000,  0.0000,  0.0000,  0.0000,  0.0000,  0.0000,  0.0000],
          [ 0.0000,  0.0000, -0.1678, -0.4418,  1.9466,  0.0000,  0.0000],
          [ 0.0000,  0.0000,  0.9604, -0.4219, -0.5241,  0.0000,  0.0000],
          [ 0.0000,  0.0000, -0.9162, -0.5436, -0.6446,  0.0000,  0.0000],
          [ 0.0000,  0.0000,  0.0000,  0.0000,  0.0000,  0.0000,  0.0000],
          [ 0.0000,  0.0000,  0.0000,  0.0000,  0.0000,  0.0000,  0.0000]]]])
>>> # using different paddings for different sides
>>> m = nn.ZeroPad2d((1, 1, 2, 0))
>>> m(input)
tensor([[[[ 0.0000,  0.0000,  0.0000,  0.0000,  0.0000],
          [ 0.0000,  0.0000,  0.0000,  0.0000,  0.0000],
          [ 0.0000, -0.1678, -0.4418,  1.9466,  0.0000],
          [ 0.0000,  0.9604, -0.4219, -0.5241,  0.0000],
          [ 0.0000, -0.9162, -0.5436, -0.6446,  0.0000]]]])
```

16.5.7 ConstantPad1d

class torch.nn.**ConstantPad1d**(*padding*, *value*)

Pads the input tensor boundaries with a constant value.

For *N*-dimensional padding, use *torch.nn.functional.pad()*.

Parameters **padding** (*int*, *tuple*) – the size of the padding. If is *int*, uses the same padding in both boundaries. If a 2-*tuple*, uses (padding_left, padding_right)

Shape:

- Input: (N, C, W_{in})
- Output: (N, C, W_{out}) where

  $W_{out} = W_{in} + \text{padding\_left} + \text{padding\_right}$

Examples:

```
>>> m = nn.ConstantPad1d(2, 3.5)
>>> input = torch.randn(1, 2, 4)
>>> input
tensor([[[-1.0491, -0.7152, -0.0749,  0.8530],
         [-1.3287,  1.8966,  0.1466, -0.2771]]])
>>> m(input)
tensor([[[ 3.5000,  3.5000, -1.0491, -0.7152, -0.0749,  0.8530,  3.5000,
           3.5000],
         [ 3.5000,  3.5000, -1.3287,  1.8966,  0.1466, -0.2771,  3.5000,
           3.5000]]])
>>> m = nn.ConstantPad1d(2, 3.5)
>>> input = torch.randn(1, 2, 3)
>>> input
tensor([[[ 1.6616,  1.4523, -1.1255],
         [-3.6372,  0.1182, -1.8652]]])
>>> m(input)
tensor([[[ 3.5000,  3.5000,  1.6616,  1.4523, -1.1255,  3.5000,  3.5000],
         [ 3.5000,  3.5000, -3.6372,  0.1182, -1.8652,  3.5000,  3.5000]]])
>>> # using different paddings for different sides
```

(continues on next page)

(continued from previous page)

```
>>> m = nn.ConstantPad1d((3, 1), 3.5)
>>> m(input)
tensor([[[ 3.5000,  3.5000,  3.5000,  1.6616,  1.4523, -1.1255,  3.5000],
         [ 3.5000,  3.5000,  3.5000, -3.6372,  0.1182, -1.8652,  3.5000]]])
```

16.5.8 ConstantPad2d

class torch.nn.**ConstantPad2d**(*padding*, *value*)

Pads the input tensor boundaries with a constant value.

For *N*-dimensional padding, use *torch.nn.functional.pad()*.

Parameters **padding** (*int, tuple*) – the size of the padding. If is *int*, uses the same padding in all boundaries. If a 4-*tuple*, uses (padding_left, padding_right, padding_top, padding_bottom)

Shape:

- Input: (N, C, H_{in}, W_{in})
- Output: (N, C, H_{out}, W_{out}) where

  $H_{out} = H_{in} + \text{padding\_top} + \text{padding\_bottom}$

  $W_{out} = W_{in} + \text{padding\_left} + \text{padding\_right}$

Examples:

```
>>> m = nn.ConstantPad2d(2, 3.5)
>>> input = torch.randn(1, 2, 2)
>>> input
tensor([[[ 1.6585,  0.4320],
         [-0.8701, -0.4649]]])
>>> m(input)
tensor([[[ 3.5000,  3.5000,  3.5000,  3.5000,  3.5000,  3.5000],
         [ 3.5000,  3.5000,  3.5000,  3.5000,  3.5000,  3.5000],
         [ 3.5000,  3.5000,  1.6585,  0.4320,  3.5000,  3.5000],
         [ 3.5000,  3.5000, -0.8701, -0.4649,  3.5000,  3.5000],
         [ 3.5000,  3.5000,  3.5000,  3.5000,  3.5000,  3.5000],
         [ 3.5000,  3.5000,  3.5000,  3.5000,  3.5000,  3.5000]]])
>>> # using different paddings for different sides
>>> m = nn.ConstantPad2d((3, 0, 2, 1), 3.5)
>>> m(input)
tensor([[[ 3.5000,  3.5000,  3.5000,  3.5000,  3.5000],
         [ 3.5000,  3.5000,  3.5000,  3.5000,  3.5000],
         [ 3.5000,  3.5000,  3.5000,  1.6585,  0.4320],
         [ 3.5000,  3.5000,  3.5000, -0.8701, -0.4649],
         [ 3.5000,  3.5000,  3.5000,  3.5000,  3.5000]]])
```

16.5.9 ConstantPad3d

class torch.nn.**ConstantPad3d**(*padding*, *value*)

Pads the input tensor boundaries with a constant value.

For *N*-dimensional padding, use *torch.nn.functional.pad()*.

Parameters **padding** (*int, tuple*) – the size of the padding. If is *int*, uses the same padding in all boundaries. If a 6-*tuple*, uses (padding_left, padding_right, padding_top, padding_bottom, padding_front, padding_back)

Shape:

- Input: $(N, C, D_{in}, H_{in}, W_{in})$
- Output: $(N, C, D_{out}, H_{out}, W_{out})$ where

  $D_{out} = D_{in} + \text{padding\_front} + \text{padding\_back}$

  $H_{out} = H_{in} + \text{padding\_top} + \text{padding\_bottom}$

  $W_{out} = W_{in} + \text{padding\_left} + \text{padding\_right}$

Examples:

```
>>> m = nn.ConstantPad3d(3, 3.5)
>>> input = torch.randn(16, 3, 10, 20, 30)
>>> output = m(input)
>>> # using different paddings for different sides
>>> m = nn.ConstantPad3d((3, 3, 6, 6, 0, 1), 3.5)
>>> output = m(input)
```

16.6 Non-linear activations (weighted sum, nonlinearity)

16.6.1 ELU

class torch.nn.**ELU**(*alpha=1.0*, *inplace=False*)

Applies the element-wise function:

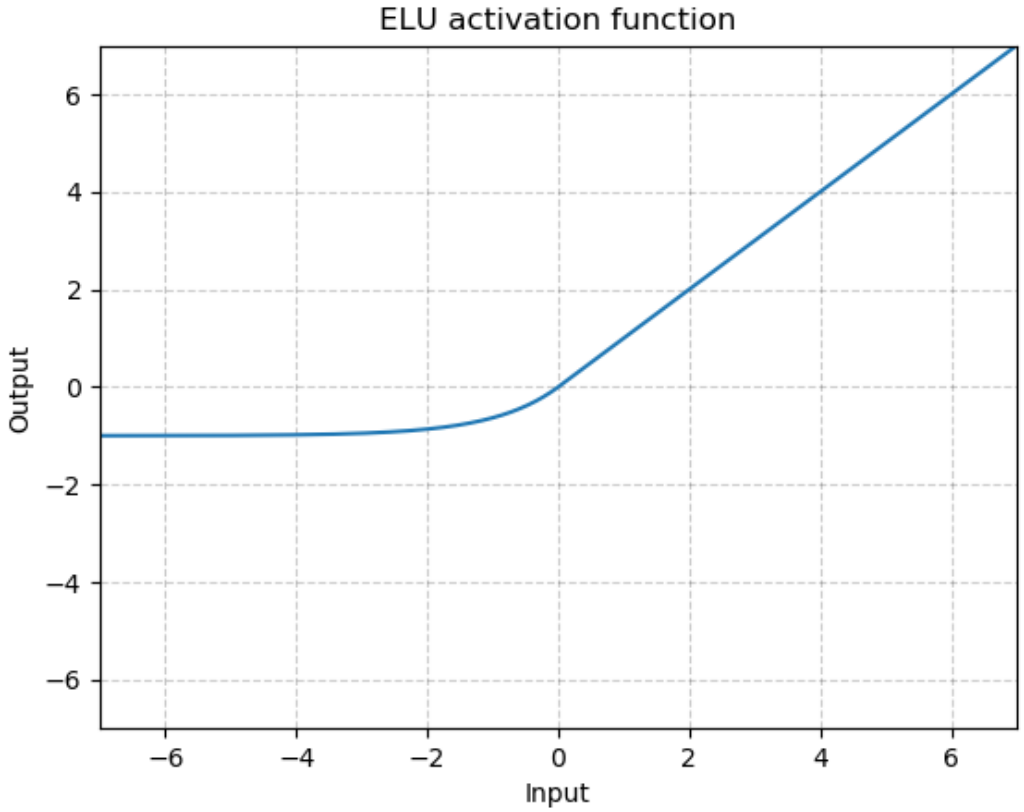
$$\text{ELU}(x) = \max(0, x) + \min(0, \alpha * (\exp(x) - 1))$$

Parameters

- **alpha** – the α value for the ELU formulation. Default: 1.0
- **inplace** – can optionally do the operation in-place. Default: False

Shape:

- Input: $(N, *)$ where * means, any number of additional dimensions
- Output: $(N, *)$, same shape as the input



Examples:

```
>>> m = nn.ELU()
>>> input = torch.randn(2)
>>> output = m(input)
```

16.6.2 Hardshrink

class torch.nn.**Hardshrink**(*lambd=0.5*)

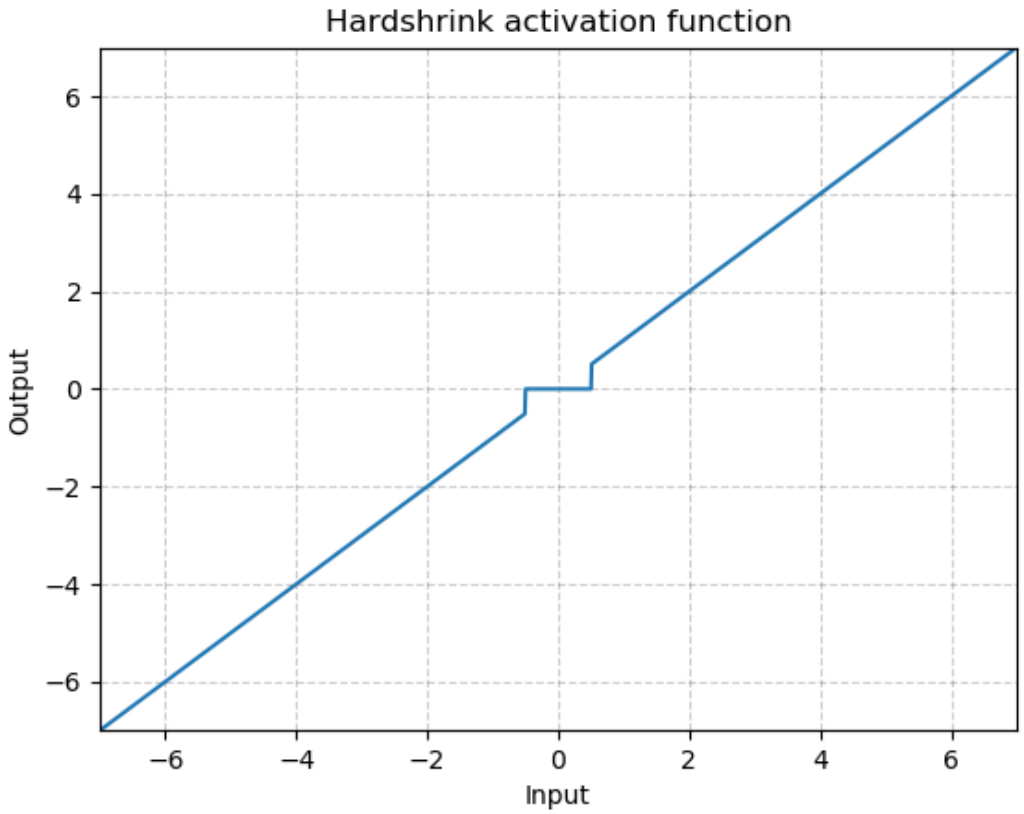
Applies the hard shrinkage function element-wise:

$$\text{HardShrink}(x) = \begin{cases} x, & \text{if } x > \lambda \\ x, & \text{if } x < -\lambda \\ 0, & \text{otherwise} \end{cases}$$

Parameters **lambd** – the λ value for the Hardshrink formulation. Default: 0.5

Shape:

- Input: $(N, *)$ where * means, any number of additional dimensions
- Output: $(N, *)$, same shape as the input



Examples:

```
>>> m = nn.Hardshrink()
>>> input = torch.randn(2)
>>> output = m(input)
```

16.6.3 Hardtanh

class torch.nn.**Hardtanh**(*min_val=-1.0*, *max_val=1.0*, *inplace=False*, *min_value=None*, *max_value=None*)

Applies the HardTanh function element-wise

HardTanh is defined as:

$$\text{HardTanh}(x) = \begin{cases} 1 & \text{if } x > 1 \\ -1 & \text{if } x < -1 \\ x & \text{otherwise} \end{cases}$$

The range of the linear region $[-1, 1]$ can be adjusted using `min_val` and `max_val`.

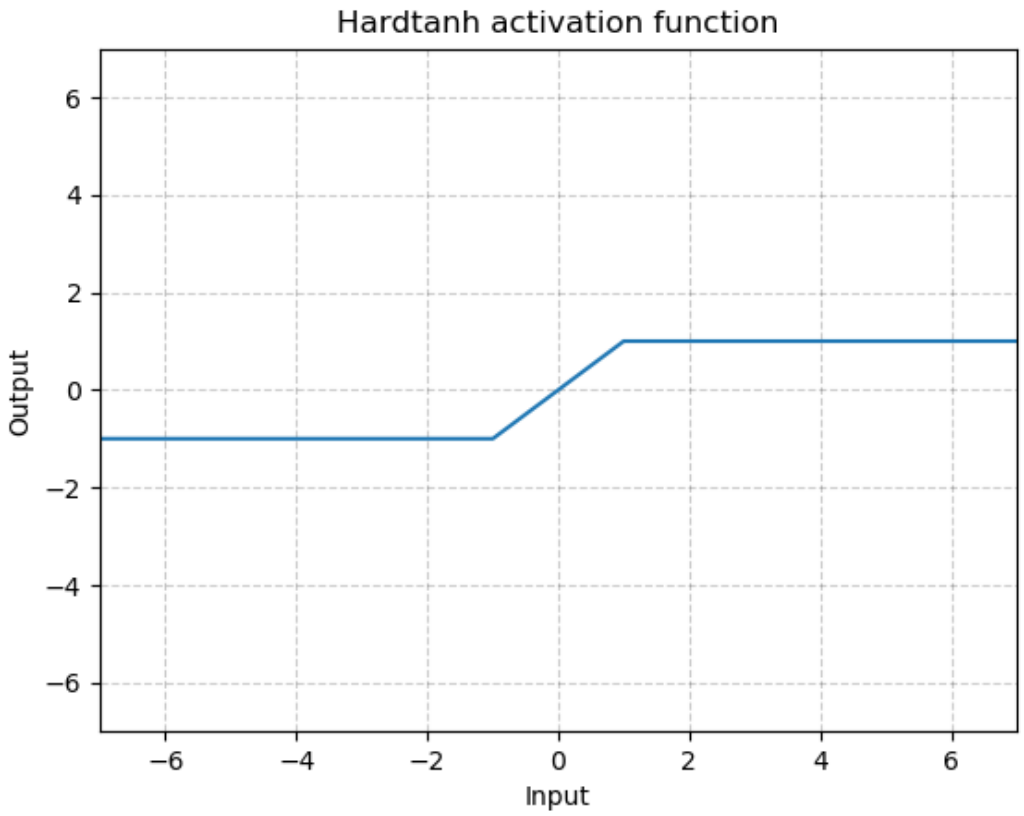
Parameters

- **min_val** – minimum value of the linear region range. Default: -1
- **max_val** – maximum value of the linear region range. Default: 1
- **inplace** – can optionally do the operation in-place. Default: `False`

Keyword arguments `min_value` and `max_value` have been deprecated in favor of `min_val` and `max_val`.

Shape:

- Input: $(N, *)$ where * means, any number of additional dimensions
- Output: $(N, *)$, same shape as the input



Examples:

```
>>> m = nn.Hardtanh(-2, 2)
>>> input = torch.randn(2)
>>> output = m(input)
```

16.6.4 LeakyReLU

class `torch.nn.`**LeakyReLU**(*negative_slope=0.01*, *inplace=False*)

Applies the element-wise function:

$$\text{LeakyReLU}(x) = \max(0, x) + \text{negative\_slope} * \min(0, x)$$

or

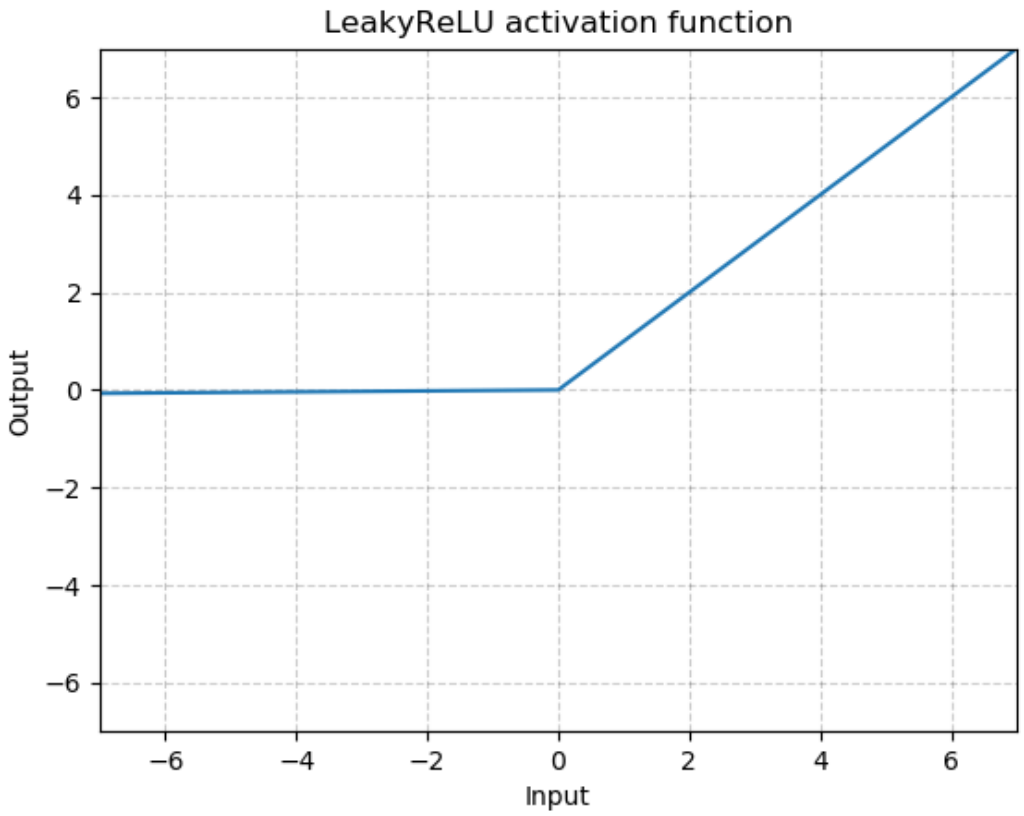
$$\text{LeakyRELU}(x) = \begin{cases} x, & \text{if } x \geq 0 \\ \text{negative\_slope} \times x, & \text{otherwise} \end{cases}$$

Parameters

- **negative_slope** – Controls the angle of the negative slope. Default: 1e-2
- **inplace** – can optionally do the operation in-place. Default: `False`

Shape:

- Input: $(N, *)$ where * means, any number of additional dimensions
- Output: $(N, *)$, same shape as the input



Examples:

```
>>> m = nn.LeakyReLU(0.1)
>>> input = torch.randn(2)
>>> output = m(input)
```

16.6.5 LogSigmoid

class torch.nn.**LogSigmoid**

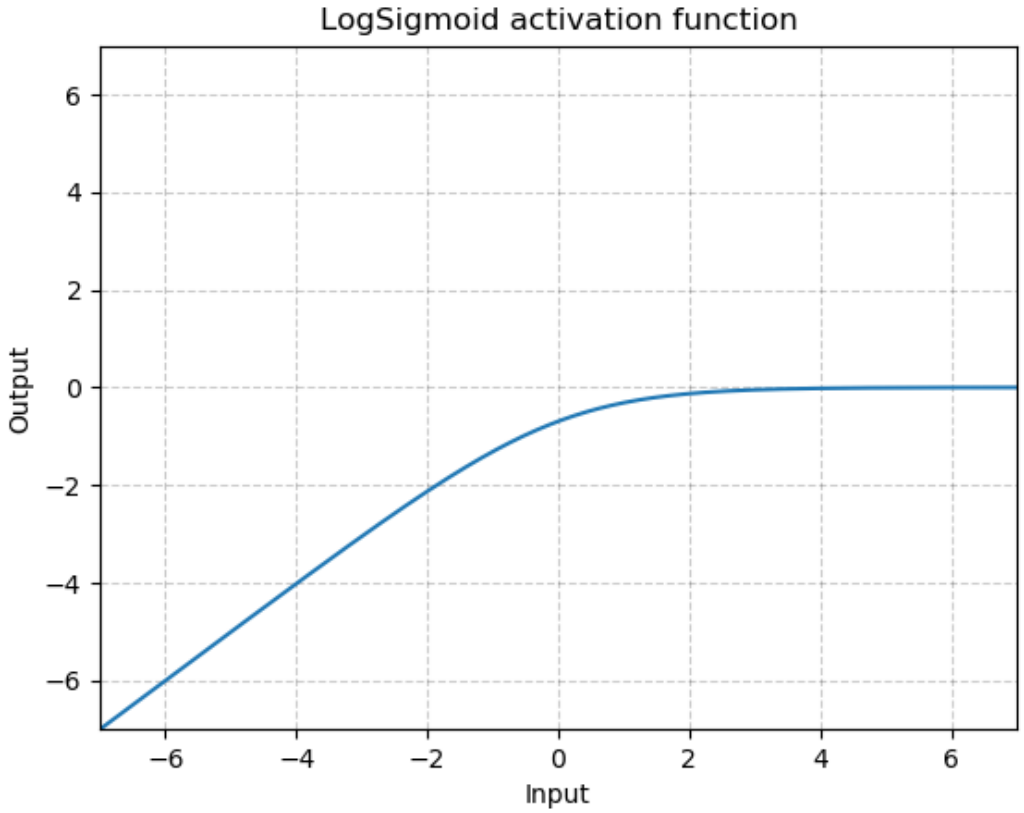
Applies the element-wise function:

$$\text{LogSigmoid}(x) = \log\left(\frac{1}{1 + \exp(-x)}\right)$$

Shape:

- Input: $(N, *)$ where * means, any number of additional dimensions

- Output: $(N, *)$, same shape as the input



Examples:

```
>>> m = nn.LogSigmoid()
>>> input = torch.randn(2)
>>> output = m(input)
```

16.6.6 MultiheadAttention

16.6.7 PReLU

class torch.nn.**PReLU**(*num_parameters=1*, *init=0.25*)

Applies the element-wise function:

$$\text{PReLU}(x) = \max(0, x) + a * \min(0, x)$$

or

$$\text{PReLU}(x) = \begin{cases} x, & \text{if } x \geq 0 \\ ax, & \text{otherwise} \end{cases}$$

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.

Note: weight decay should not be used when learning a for good performance.

Note: Channel dim is the 2nd dim of input. When input has dims < 2, then there is no channel dim and the number of channels = 1.

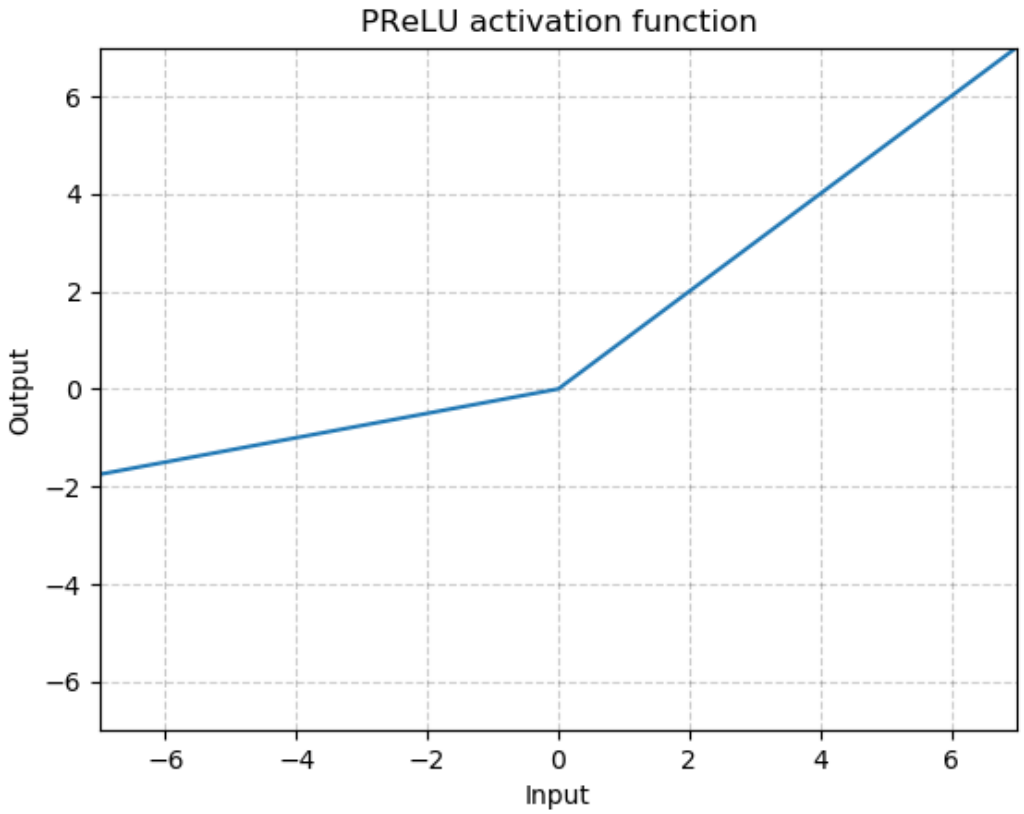
Parameters

- **num_parameters** (*int*) – number of a to learn. Although it takes an int as input, there is only two values are legitimate: 1, or the number of channels at input. Default: 1
- **init** (*float*) – 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

Variables **weight** (Tensor) – the learnable weights of shape (num_parameters).



Examples:

```
>>> m = nn.PReLU()
>>> input = torch.randn(2)
>>> output = m(input)
```

16.6.8 ReLU

class torch.nn.**ReLU**(*inplace=False*)

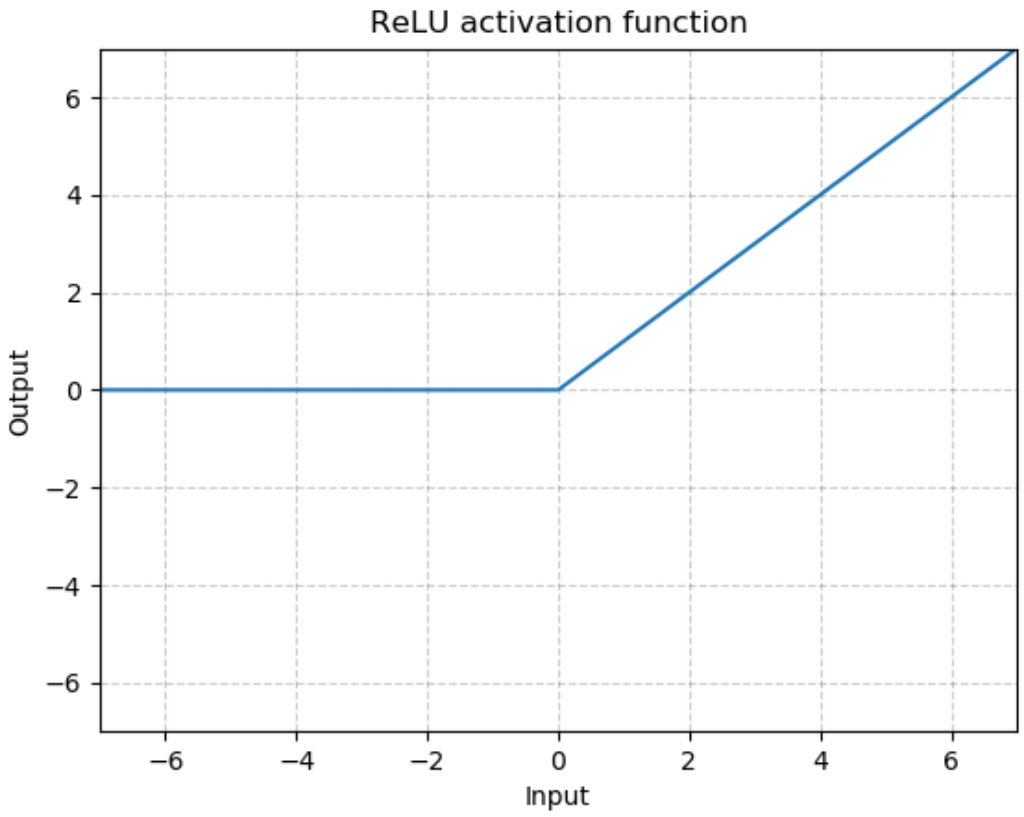
Applies the rectified linear unit function element-wise:

$\text{ReLU}(x) = \max(0, x)$

Parameters **inplace** – can optionally do the operation in-place. Default: `False`

Shape:

- Input: $(N, *)$ where * means, any number of additional dimensions
- Output: $(N, *)$, same shape as the input



Examples:

```
>>> m = nn.ReLU()
>>> input = torch.randn(2)
>>> output = m(input)
```

(continues on next page)

(continued from previous page)

```
An implementation of CReLU - https://arxiv.org/abs/1603.05201

  >>> m = nn.ReLU()
  >>> input = torch.randn(2).unsqueeze(0)
  >>> output = torch.cat((m(input),m(-input)))
```

16.6.9 ReLU6

class torch.nn.**ReLU6**(*inplace=False*)

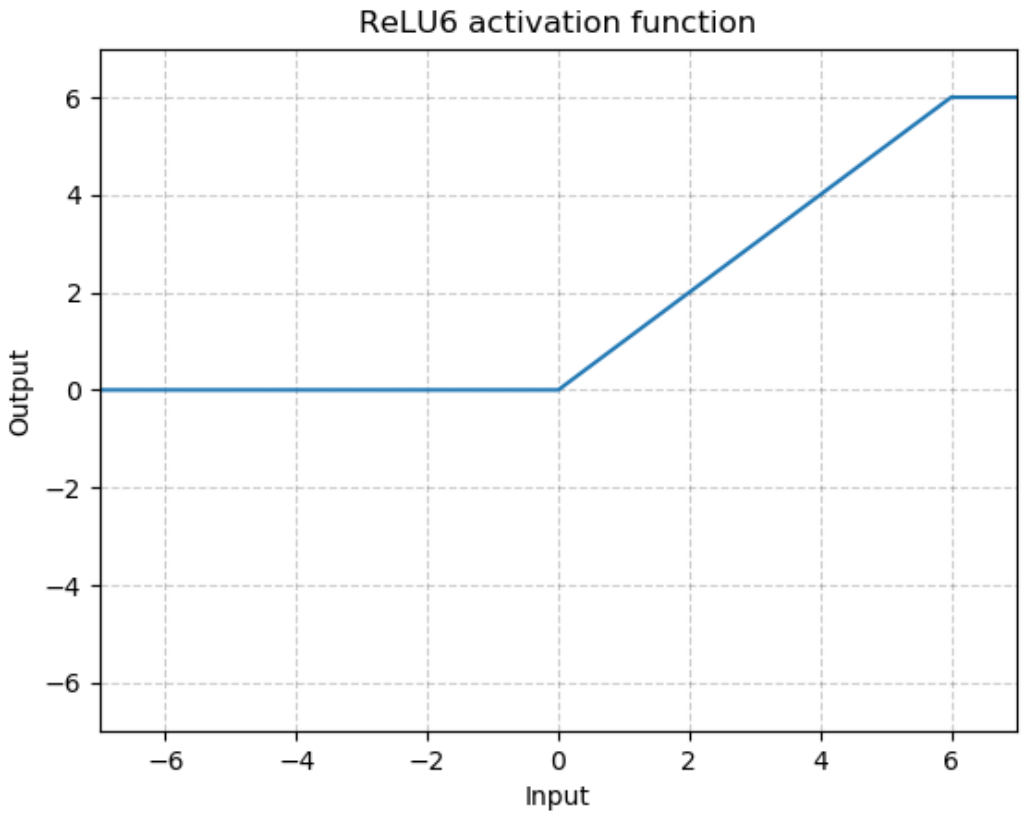
Applies the element-wise function:

$$\text{ReLU6}(x) = \min(\max(0, x), 6)$$

Parameters **inplace** – can optionally do the operation in-place. Default: False

Shape:

- Input: $(N, *)$ where * means, any number of additional dimensions
- Output: $(N, *)$, same shape as the input



Examples:

```
>>> m = nn.ReLU6()
>>> input = torch.randn(2)
>>> output = m(input)
```

16.6.10 RReLU

class torch.nn.**RReLU**(*lower=0.125*, *upper=0.3333333333333333*, *inplace=False*)

Applies the randomized leaky rectified liner unit function, element-wise, as described in the paper:

Empirical Evaluation of Rectified Activations in Convolutional Network.

The function is defined as:

$$\text{RReLU}(x) = \begin{cases} x & \text{if } x \geq 0 \\ ax & \text{otherwise} \end{cases}$$

where a is randomly sampled from uniform distribution $\mathcal{U}(\text{lower}, \text{upper})$.

See: https://arxiv.org/pdf/1505.00853.pdf

Parameters

- **lower** – lower bound of the uniform distribution. Default: $\frac{1}{8}$
- **upper** – upper bound of the uniform distribution. Default: $\frac{1}{3}$
- **inplace** – can optionally do the operation in-place. Default: `False`

Shape:

- Input: $(N, *)$ where * means, any number of additional dimensions
- Output: $(N, *)$, same shape as the input

Examples:

```
>>> m = nn.RReLU(0.1, 0.3)
>>> input = torch.randn(2)
>>> output = m(input)
```

16.6.11 SELU

class torch.nn.**SELU**(*inplace=False*)

Applied element-wise, as:

$$\text{SELU}(x) = \text{scale} * (\max(0, x) + \min(0, \alpha * (\exp(x) - 1)))$$

with $\alpha = 1.6732632423543772848170429916717$ and scale $= 1.0507009873554804934193349852946$.

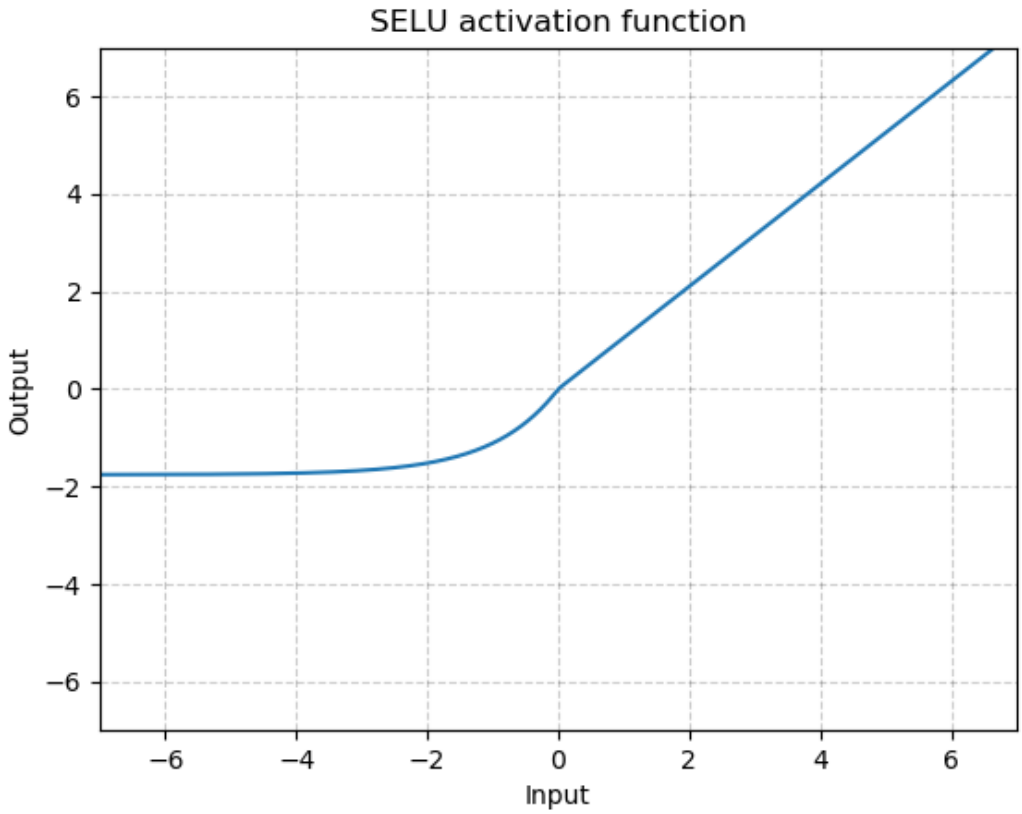
More details can be found in the paper Self-Normalizing Neural Networks .

Parameters **inplace** (*bool, optional*) – can optionally do the operation in-place. Default: `False`

Shape:

- Input: $(N, *)$ where * means, any number of additional dimensions

- Output: $(N, *)$, same shape as the input



Examples:

```
>>> m = nn.SELU()
>>> input = torch.randn(2)
>>> output = m(input)
```

16.6.12 CELU

class torch.nn.**CELU**(*alpha=1.0*, *inplace=False*)

Applies the element-wise function:

$$\text{CELU}(x) = \max(0, x) + \min(0, \alpha * (\exp(x/\alpha) - 1))$$

More details can be found in the paper Continuously Differentiable Exponential Linear Units .

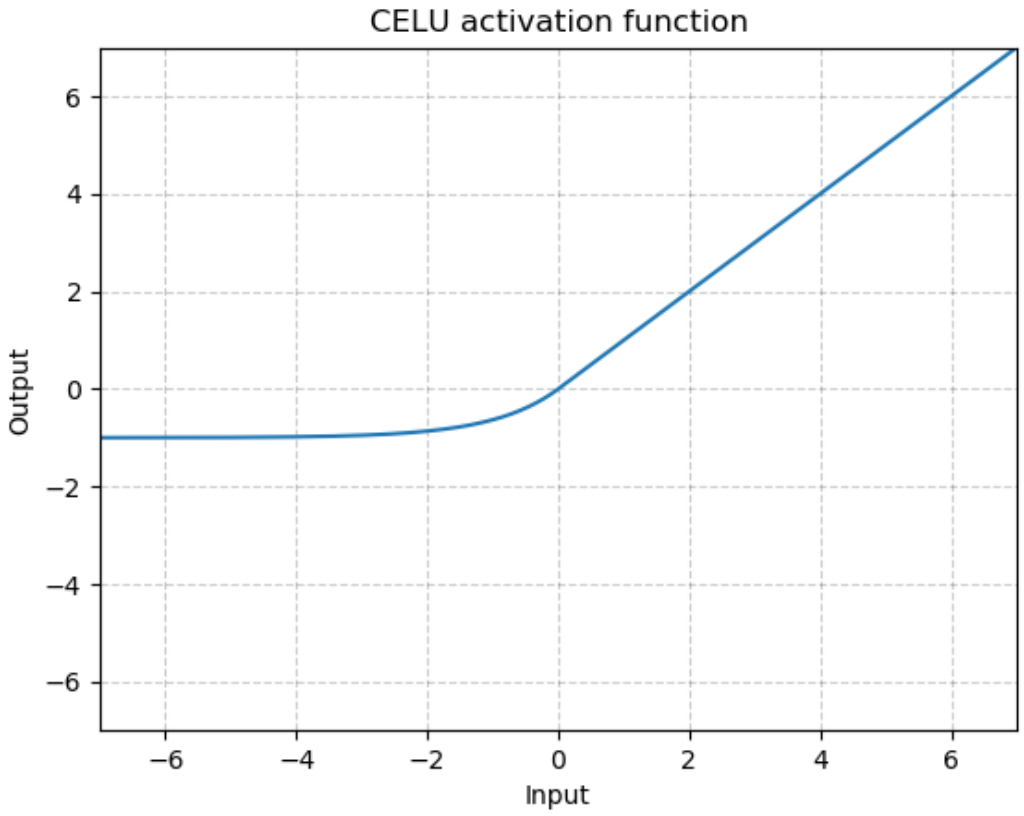
Parameters

- **alpha** – the α value for the CELU formulation. Default: 1.0
- **inplace** – can optionally do the operation in-place. Default: `False`

Shape:

- Input: $(N, *)$ where * means, any number of additional dimensions

- Output: $(N, *)$, same shape as the input



Examples:

```
>>> m = nn.CELU()
>>> input = torch.randn(2)
>>> output = m(input)
```

16.6.13 Sigmoid

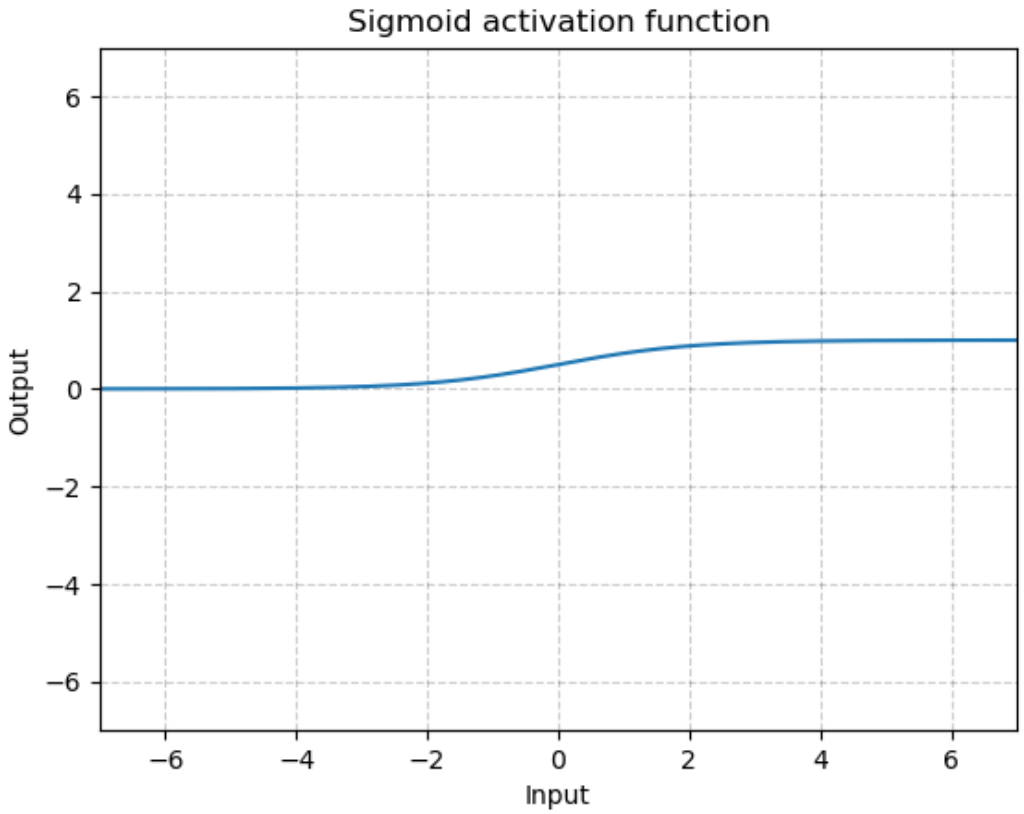
class torch.nn.**Sigmoid**

Applies the element-wise function:

$$\text{Sigmoid}(x) = \frac{1}{1 + \exp(-x)}$$

Shape:

- Input: $(N, *)$ where * means, any number of additional dimensions
- Output: $(N, *)$, same shape as the input



Examples:

```
>>> m = nn.Sigmoid()
>>> input = torch.randn(2)
>>> output = m(input)
```

16.6.14 Softplus

class torch.nn.**Softplus**(*beta=1*, *threshold=20*)

Applies the element-wise function:

$$\text{Softplus}(x) = \frac{1}{\beta} * \log(1 + \exp(\beta * x))$$

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.

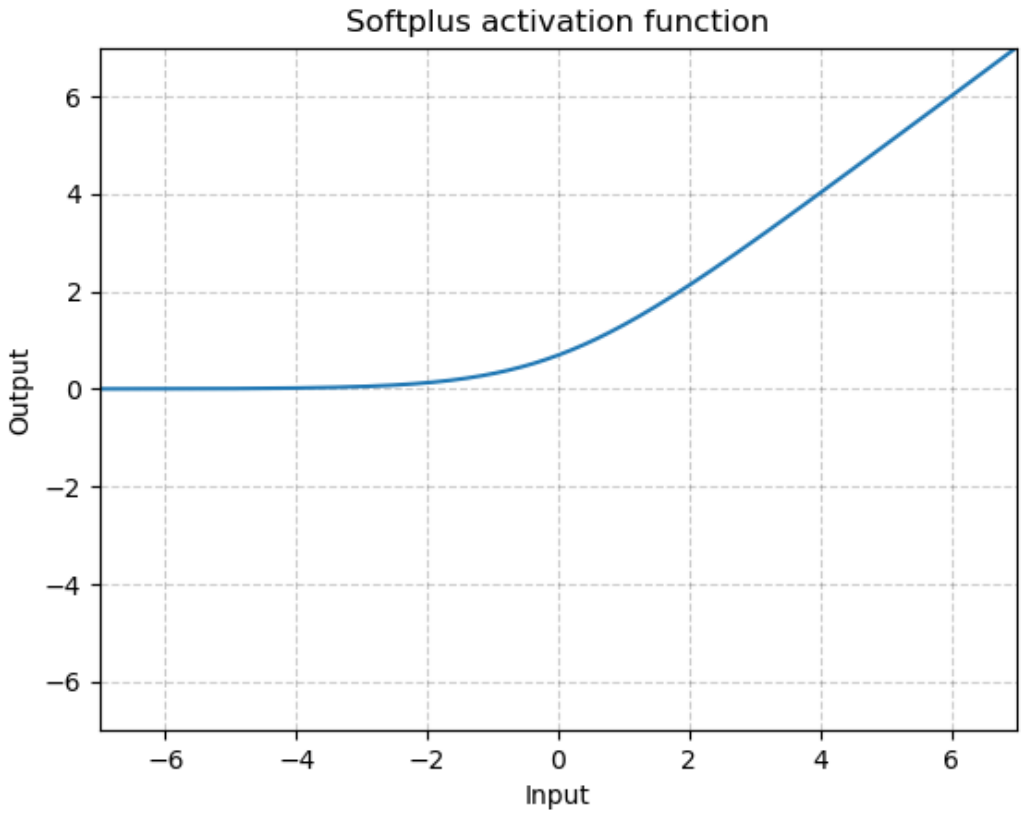
Parameters

- **beta** – the β 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 = torch.randn(2)
>>> output = m(input)
```

16.6.15 Softshrink

class torch.nn.**Softshrink**(*lambd=0.5*)

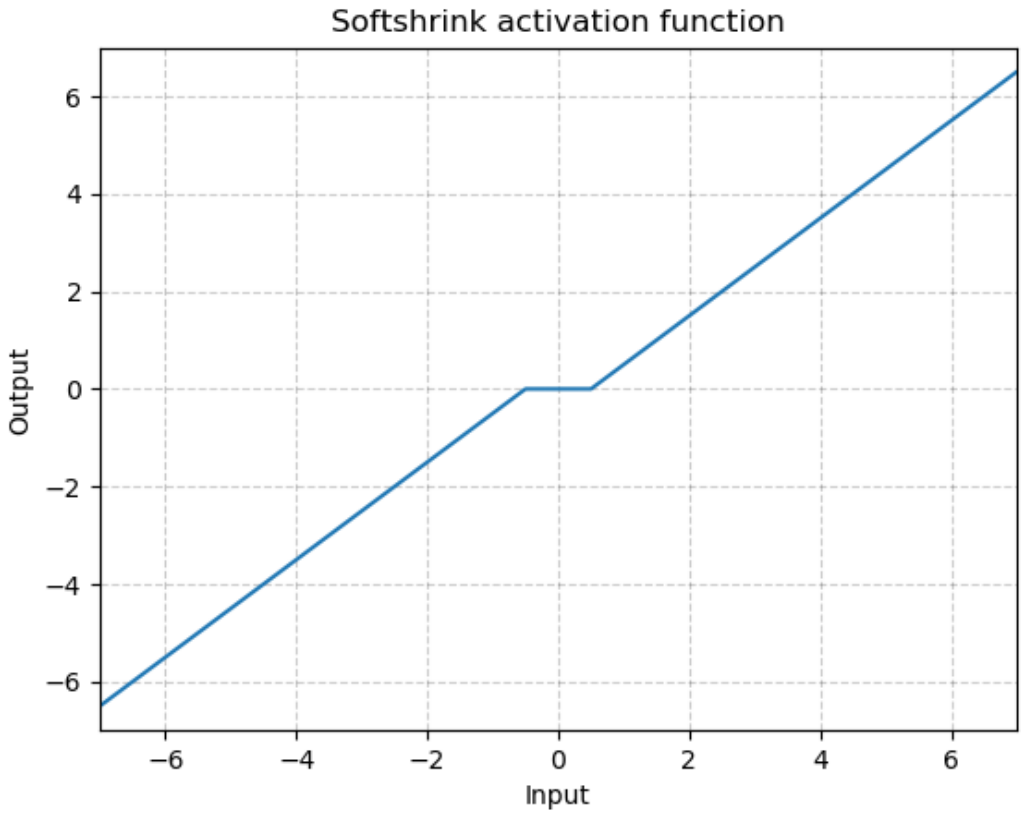
Applies the soft shrinkage function elementwise:

$$\text{SoftShrinkage}(x) = \begin{cases} x - \lambda, & \text{if } x > \lambda \\ x + \lambda, & \text{if } x < -\lambda \\ 0, & \text{otherwise} \end{cases}$$

Parameters **lambd** – the λ 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 = torch.randn(2)
>>> output = m(input)
```

16.6.16 Softsign

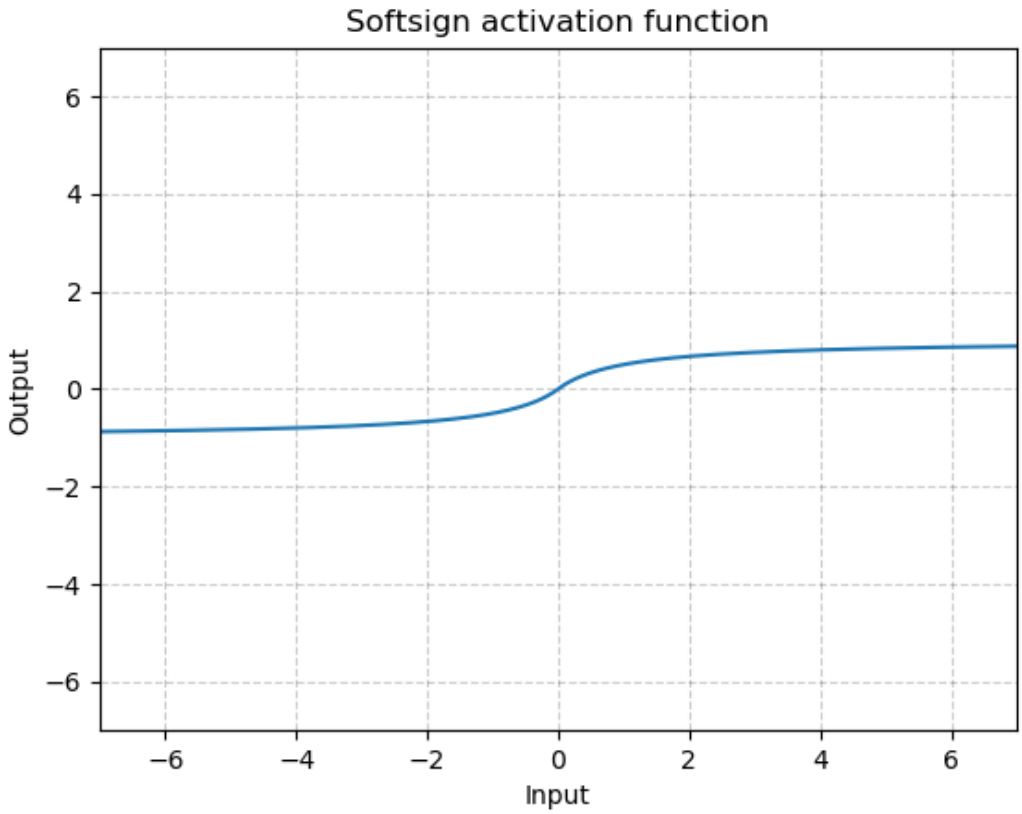
class torch.nn.**Softsign**

Applies the element-wise function:

$$\text{SoftSign}(x) = \frac{x}{1 + |x|}$$

Shape:

- Input: $(N, *)$ where * means, any number of additional dimensions
- Output: $(N, *)$, same shape as the input



Examples:

```
>>> m = nn.Softsign()
>>> input = torch.randn(2)
>>> output = m(input)
```

16.6.17 Tanh

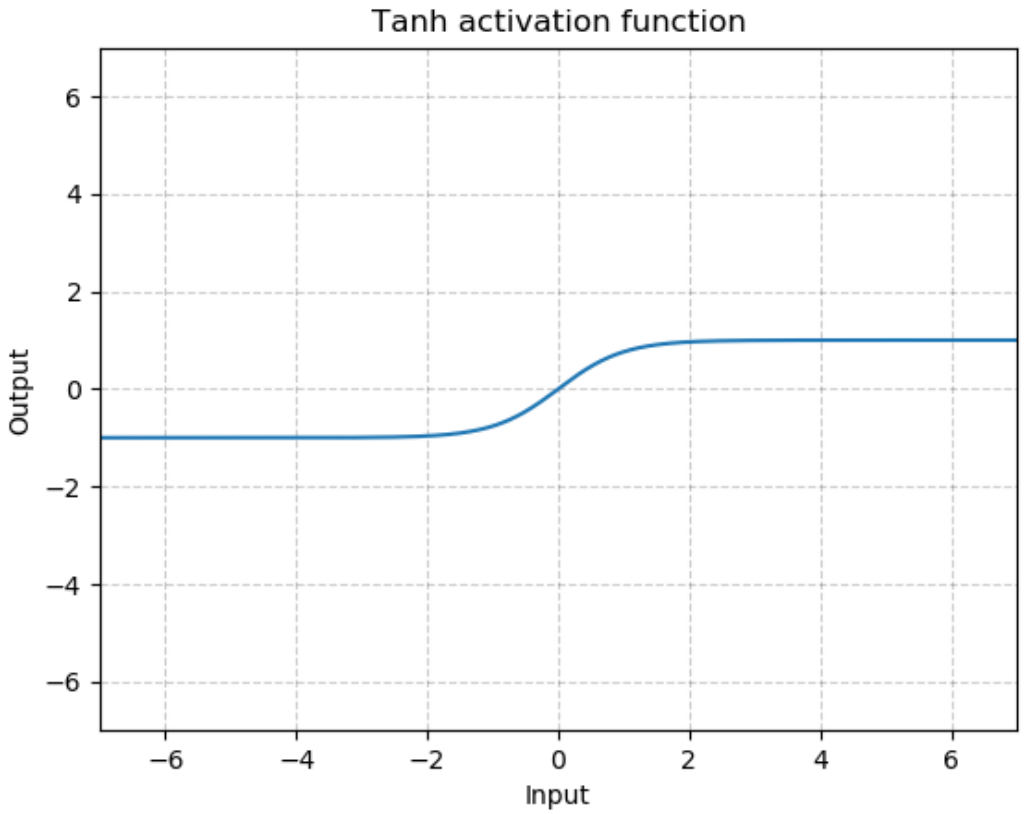
class torch.nn.**Tanh**

Applies the element-wise function:

$$\text{Tanh}(x) = \tanh(x) = \frac{e^x - e^{-x}}{e^x + e^{-x}}$$

Shape:

- Input: $(N, *)$ where * means, any number of additional dimensions
- Output: $(N, *)$, same shape as the input



Examples:

```
>>> m = nn.Tanh()
>>> input = torch.randn(2)
>>> output = m(input)
```

16.6.18 Tanhshrink

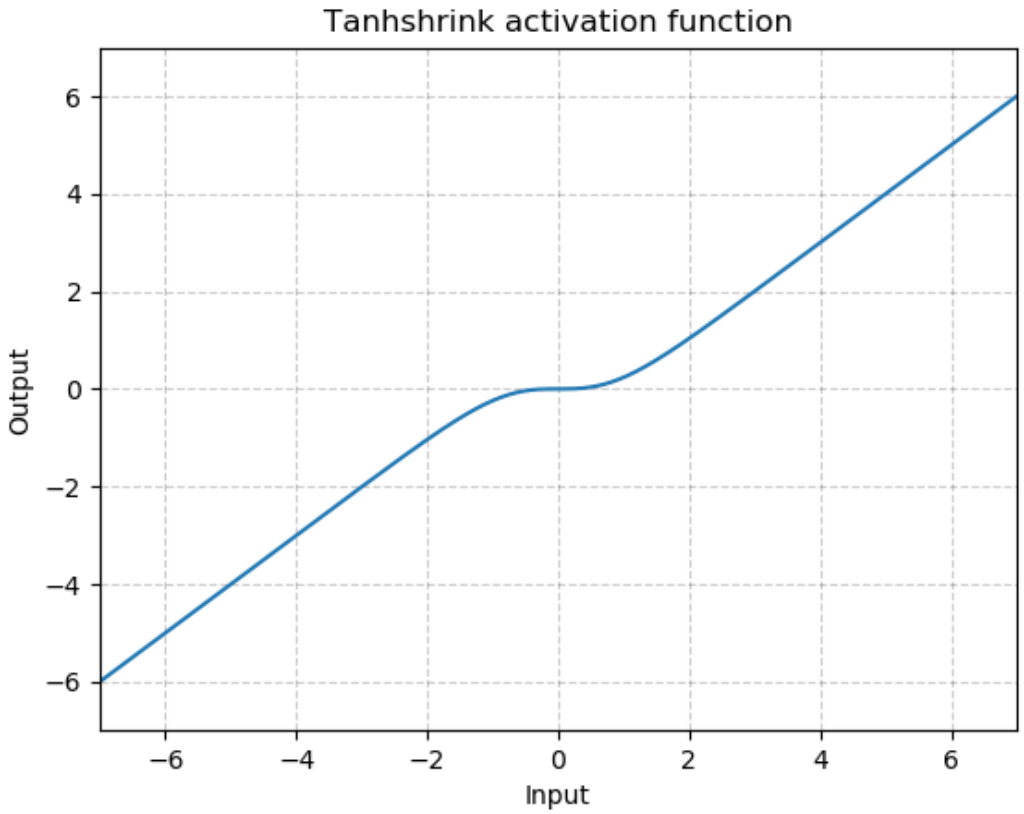
class torch.nn.**Tanhshrink**

Applies the element-wise function:

$$\text{Tanhshrink}(x) = x - \text{Tanh}(x)$$

Shape:

- Input: $(N, *)$ where * means, any number of additional dimensions
- Output: $(N, *)$, same shape as the input



Examples:

```
>>> m = nn.Tanhshrink()
>>> input = torch.randn(2)
>>> output = m(input)
```

16.6.19 Threshold

class torch.nn.**Threshold**(*threshold*, *value*, *inplace=False*)

Thresholds each element of the input Tensor.

Threshold is defined as:

$$y = \begin{cases} x, & \text{if } x > \text{threshold} \\ \text{value}, & \text{otherwise} \end{cases}$$

Parameters

- **threshold** – The value to threshold at
- **value** – The value to replace with
- **inplace** – can optionally do the operation in-place. Default: `False`

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 = torch.randn(2)
>>> output = m(input)
```

16.7 Non-linear activations (other)

16.7.1 Softmin

class torch.nn.**Softmin**(*dim=None*)

Applies the Softmin function to an n-dimensional input Tensor rescaling them so that the elements of the n-dimensional output Tensor lie in the range *[0, 1]* and sum to 1.

Softmin is defined as:

$$\text{Softmin}(x_i) = \frac{\exp(-x_i)}{\sum_j \exp(-x_j)}$$

Shape:

- Input: $(*)$ where * means, any number of additional dimensions
- Output: $(*)$, same shape as the input

Parameters **dim** (*int*) – A dimension along which Softmin will be computed (so every slice along dim will sum to 1).

Returns a Tensor of the same dimension and shape as the input, with values in the range [0, 1]

Examples:

```
>>> m = nn.Softmin()
>>> input = torch.randn(2, 3)
>>> output = m(input)
```

16.7.2 Softmax

class torch.nn.**Softmax**(*dim=None*)

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:

$$\text{Softmax}(x_i) = \frac{\exp(x_i)}{\sum_j \exp(x_j)}$$

Shape:

- Input: $(*)$ where * means, any number of additional dimensions
- Output: $(*)$, same shape as the input

Returns a Tensor of the same dimension and shape as the input with values in the range [0, 1]

Parameters **dim** (*int*) – A dimension along which Softmax will be computed (so every slice along dim will sum to 1).

Note: This module doesnt work directly with NLLLoss, which expects the Log to be computed between the Softmax and itself. Use *LogSoftmax* instead (its faster and has better numerical properties).

Examples:

```
>>> m = nn.Softmax()
>>> input = torch.randn(2, 3)
>>> output = m(input)
```

16.7.3 Softmax2d

class torch.nn.**Softmax2d**

Applies SoftMax over features to each spatial location.

When given an image of `Channels x Height x Width`, it will apply *Softmax* to each location $(Channels, h_i, w_j)$

Shape:

- Input: (N, C, H, W)
- Output: (N, C, H, W) (same shape as input)

Returns a Tensor of the same dimension and shape as the input with values in the range [0, 1]

Examples:

```
>>> m = nn.Softmax2d()
>>> # you softmax over the 2nd dimension
>>> input = torch.randn(2, 3, 12, 13)
>>> output = m(input)
```

16.7.4 LogSoftmax

class torch.nn.**LogSoftmax**(*dim=None*)

Applies the $\log(\text{Softmax}(x))$ function to an n-dimensional input Tensor. The LogSoftmax formulation can be simplified as:

$$\text{LogSoftmax}(x_i) = \log\left(\frac{\exp(x_i)}{\sum_j \exp(x_j)}\right)$$

Shape:

- Input: $(*)$ where * means, any number of additional dimensions
- Output: $(*)$, same shape as the input

Parameters **dim** (*int*) – A dimension along which LogSoftmax will be computed.

Returns a Tensor of the same dimension and shape as the input with values in the range [-inf, 0)

Examples:

```
>>> m = nn.LogSoftmax()
>>> input = torch.randn(2, 3)
>>> output = m(input)
```

16.7.5 AdaptiveLogSoftmaxWithLoss

class torch.nn.**AdaptiveLogSoftmaxWithLoss**(*in_features*, *n_classes*, *cutoffs*, *div_value=4.0*, *head_bias=False*)

Efficient softmax approximation as described in Efficient softmax approximation for GPUs by Edouard Grave, Armand Joulin, Moustapha Cissé, David Grangier, and Hervé Jégou.

Adaptive softmax is an approximate strategy for training models with large output spaces. It is most effective when the label distribution is highly imbalanced, for example in natural language modelling, where the word frequency distribution approximately follows the Zipfs law.

Adaptive softmax partitions the labels into several clusters, according to their frequency. These clusters may contain different number of targets each. Additionally, clusters containing less frequent labels assign lower dimensional embeddings to those labels, which speeds up the computation. For each minibatch, only clusters for which at least one target is present are evaluated.

The idea is that the clusters which are accessed frequently (like the first one, containing most frequent labels), should also be cheap to compute – that is, contain a small number of assigned labels.

We highly recommend taking a look at the original paper for more details.

- `cutoffs` should be an ordered Sequence of integers sorted in the increasing order. It controls number of clusters and the partitioning of targets into clusters. For example setting `cutoffs = [10, 100, 1000]` means that first *10* targets will be assigned to the head of the adaptive softmax, targets *11, 12, , 100* will be assigned to the first cluster, and targets *101, 102, , 1000* will be assigned to the second cluster, while targets *1001, 1002, , n_classes - 1* will be assigned to the last, third cluster.
- `div_value` is used to compute the size of each additional cluster, which is given as $\left\lfloor \frac{in\_features}{div\_value^{idx}} \right\rfloor$, where idx is the cluster index (with clusters for less frequent words having larger indices, and indices starting from 1).
- `head_bias` if set to True, adds a bias term to the head of the adaptive softmax. See paper for details. Set to False in the official implementation.

Warning: Labels passed as inputs to this module should be sorted accoridng to their frequency. This means that the most frequent label should be represented by the index *0*, and the least frequent label should be represented by the index *n_classes - 1*.

Note: This module returns a `NamedTuple` with `output` and `loss` fields. See further documentation for details.

Note: To compute log-probabilities for all classes, the `log_prob` method can be used.

Parameters

- **in_features** (*int*) – Number of features in the input tensor

- **n_classes** (*int*) – Number of classes in the dataset
- **cutoffs** (*Sequence*) – Cutoffs used to assign targets to their buckets
- **div_value** (*float, optional*) – value used as an exponent to compute sizes of the clusters. Default: 4.0
- **head_bias** (*bool, optional*) – If `True`, adds a bias term to the head of the adaptive softmax. Default: `False`

Returns

- **output** is a Tensor of size `N` containing computed target log probabilities for each example
- **loss** is a Scalar representing the computed negative log likelihood loss

Return type `NamedTuple` with `output` and `loss` fields

Shape:

- input: $(N, in\_features)$
- target: (N) where each value satisfies $0 <= target[i] <= n\_classes$
- output1: (N)
- output2: `Scalar`

log_prob(*input*)

Computes log probabilities for all $n\_classes$

Parameters **input** (`Tensor`) – a minibatch of examples

Returns log-probabilities of for each class c in range $0 <= c <= n\_classes$, where $n\_classes$ is a parameter passed to `AdaptiveLogSoftmaxWithLoss` constructor.

Shape:

- Input: $(N, in\_features)$
- Output: $(N, n\_classes)$

predict(*input*)

This is equivalent to *self.log_pob(input).argmax(dim=1)*, but is more efficient in some cases.

Parameters **input** (`Tensor`) – a minibatch of examples

Returns a class with the highest probability for each example

Return type output (*Tensor*)

Shape:

- Input: $(N, in\_features)$
- Output: (N)

16.8 Normalization layers

16.8.1 BatchNorm1d

class torch.nn.**BatchNorm1d**(*num_features*, *eps=1e-05*, *momentum=0.1*, *affine=True*, *track_running_stats=True*)

Applies Batch Normalization over a 2D or 3D input (a mini-batch of 1D inputs with optional additional channel dimension) as described in the paper Batch Normalization: Accelerating Deep Network Training by Reducing Internal Covariate Shift .

$$y = \frac{x - \mathrm{E}[x]}{\sqrt{\mathrm{Var}[x] + \epsilon}} * \gamma + \beta$$

The mean and standard-deviation are calculated per-dimension over the mini-batches and γ and β are learnable parameter vectors of size *C* (where *C* is the input size). By default, the elements of γ are sampled from $\mathcal{U}(0, 1)$ and the elements of β are set to 0.

Also by default, during training this layer keeps running estimates of its computed mean and variance, which are then used for normalization during evaluation. The running estimates are kept with a default `momentum` of 0.1.

If `track_running_stats` is set to `False`, this layer then does not keep running estimates, and batch statistics are instead used during evaluation time as well.

Note: This `momentum` argument is different from one used in optimizer classes and the conventional notion of momentum. Mathematically, the update rule for running statistics here is $\hat{x}_{\text{new}} = (1 - \text{momentum}) \times \hat{x} + \text{momentum} \times x_t$, where $\hat{x}$ is the estimated statistic and x_t is the new observed value.

Because the Batch Normalization is done over the *C* dimension, computing statistics on *(N, L)* slices, its common terminology to call this Temporal Batch Normalization.

Parameters

- **num_features** – C from an expected input of size (N, C, L) or L from input of size (N, L)
- **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. Can be set to `None` for cumulative moving average (i.e. simple average). Default: 0.1
- **affine** – a boolean value that when set to `True`, this module has learnable affine parameters. Default: `True`
- **track_running_stats** – a boolean value that when set to `True`, this module tracks the running mean and variance, and when set to `False`, this module does not track such statistics and always uses batch statistics in both training and eval modes. Default: `True`

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 = torch.randn(20, 100)
>>> output = m(input)
```

16.8.2 BatchNorm2d

class torch.nn.**BatchNorm2d**(*num_features*, *eps=1e-05*, *momentum=0.1*, *affine=True*, *track_running_stats=True*)

Applies Batch Normalization over a 4D input (a mini-batch of 2D inputs with additional channel dimension) as described in the paper Batch Normalization: Accelerating Deep Network Training by Reducing Internal Covariate Shift .

$$y = \frac{x - \mathrm{E}[x]}{\sqrt{\mathrm{Var}[x] + \epsilon}} * \gamma + \beta$$

The mean and standard-deviation are calculated per-dimension over the mini-batches and γ and β are learnable parameter vectors of size *C* (where *C* is the input size). By default, the elements of γ are sampled from $\mathcal{U}(0, 1)$ and the elements of β are set to 0.

Also by default, during training this layer keeps running estimates of its computed mean and variance, which are then used for normalization during evaluation. The running estimates are kept with a default `momentum` of 0.1.

If `track_running_stats` is set to `False`, this layer then does not keep running estimates, and batch statistics are instead used during evaluation time as well.

Note: This `momentum` argument is different from one used in optimizer classes and the conventional notion of momentum. Mathematically, the update rule for running statistics here is $\hat{x}_{\text{new}} = (1 - \text{momentum}) \times \hat{x} + \text{momentum} \times x_t$, where $\hat{x}$ is the estimated statistic and x_t is the new observed value.

Because the Batch Normalization is done over the *C* dimension, computing statistics on *(N, H, W)* slices, its common terminology to call this Spatial Batch Normalization.

Parameters

- **num_features** – C from an expected input of size (N, C, H, W)
- **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. Can be set to `None` for cumulative moving average (i.e. simple average). Default: 0.1
- **affine** – a boolean value that when set to `True`, this module has learnable affine parameters. Default: `True`
- **track_running_stats** – a boolean value that when set to `True`, this module tracks the running mean and variance, and when set to `False`, this module does not track such statistics and always uses batch statistics in both training and eval modes. Default: `True`

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 = torch.randn(20, 100, 35, 45)
>>> output = m(input)
```

16.8.3 BatchNorm3d

class torch.nn.**BatchNorm3d**(*num_features*, *eps=1e-05*, *momentum=0.1*, *affine=True*, *track_running_stats=True*)

Applies Batch Normalization over a 5D input (a mini-batch of 3D inputs with additional channel dimension) as described in the paper Batch Normalization: Accelerating Deep Network Training by Reducing Internal Covariate Shift .

$$y = \frac{x - \mathrm{E}[x]}{\sqrt{\mathrm{Var}[x] + \epsilon}} * \gamma + \beta$$

The mean and standard-deviation are calculated per-dimension over the mini-batches and γ and β are learnable parameter vectors of size *C* (where *C* is the input size). By default, the elements of γ are sampled from $\mathcal{U}(0, 1)$ and the elements of β are set to 0.

Also by default, during training this layer keeps running estimates of its computed mean and variance, which are then used for normalization during evaluation. The running estimates are kept with a default `momentum` of 0.1.

If `track_running_stats` is set to `False`, this layer then does not keep running estimates, and batch statistics are instead used during evaluation time as well.

Note: This `momentum` argument is different from one used in optimizer classes and the conventional notion of momentum. Mathematically, the update rule for running statistics here is $\hat{x}_{\text{new}} = (1 - \text{momentum}) \times \hat{x} + \text{momentum} \times x_t$, where $\hat{x}$ is the estimated statistic and x_t is the new observed value.

Because the Batch Normalization is done over the *C* dimension, computing statistics on *(N, D, H, W)* slices, its common terminology to call this Volumetric Batch Normalization or Spatio-temporal Batch Normalization.

Parameters

- **num_features** – C from an expected input of size (N, C, D, H, W)
- **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. Can be set to `None` for cumulative moving average (i.e. simple average). Default: 0.1
- **affine** – a boolean value that when set to `True`, this module has learnable affine parameters. Default: `True`
- **track_running_stats** – a boolean value that when set to `True`, this module tracks the running mean and variance, and when set to `False`, this module does not track such statistics and always uses batch statistics in both training and eval modes. Default: `True`

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 = torch.randn(20, 100, 35, 45, 10)
>>> output = m(input)
```

16.8.4 GroupNorm

class torch.nn.**GroupNorm**(*num_groups*, *num_channels*, *eps=1e-05*, *affine=True*)

Applies Group Normalization over a mini-batch of inputs as described in the paper Group Normalization .

$$y = \frac{x - \mathrm{E}[x]}{\sqrt{\mathrm{Var}[x] + \epsilon}} * \gamma + \beta$$

The input channels are separated into `num_groups` groups, each containing `num_channels / num_groups` channels. The mean and standard-deviation are calculated separately over the each group. γ and β are learnable per-channel affine transform parameter vectors of size `num_channels` if `affine` is `True`.

This layer uses statistics computed from input data in both training and evaluation modes.

Parameters

- **num_groups** (*int*) – number of groups to separate the channels into
- **num_channels** (*int*) – number of channels expected in input
- **eps** – a value added to the denominator for numerical stability. Default: 1e-5
- **affine** – a boolean value that when set to `True`, this module has learnable per-channel affine parameters initialized to ones (for weights) and zeros (for biases). Default: `True`.

Shape:

- Input: $(N, C, *)$ where C = num_channels
- Output: $(N, C, *)$ (same shape as input)

Examples:

```
>>> input = torch.randn(20, 6, 10, 10)
>>> # Separate 6 channels into 3 groups
>>> m = nn.GroupNorm(3, 6)
>>> # Separate 6 channels into 6 groups (equivalent with InstanceNorm)
>>> m = nn.GroupNorm(6, 6)
>>> # Put all 6 channels into a single group (equivalent with LayerNorm)
>>> m = nn.GroupNorm(1, 6)
>>> # Activating the module
>>> output = m(input)
```

16.8.5 SyncBatchNorm

class torch.nn.**SyncBatchNorm**(*num_features*, *eps=1e-05*, *momentum=0.1*, *affine=True*, *track_running_stats=True*, *process_group=None*)

Applies Batch Normalization over a N-Dimensional input (a mini-batch of [N-2]D inputs with additional channel dimension) as described in the paper Batch Normalization: Accelerating Deep Network Training by Reducing Internal Covariate Shift .

$$y = \frac{x - \mathrm{E}[x]}{\sqrt{\mathrm{Var}[x] + \epsilon}} * \gamma + \beta$$

The mean and standard-deviation are calculated per-dimension over all mini-batches of the same process groups. γ and β are learnable parameter vectors of size *C* (where *C* is the input size). By default, the elements of γ are sampled from $\mathcal{U}(0, 1)$ and the elements of β are set to 0.

Also by default, during training this layer keeps running estimates of its computed mean and variance, which are then used for normalization during evaluation. The running estimates are kept with a default `momentum` of 0.1.

If `track_running_stats` is set to `False`, this layer then does not keep running estimates, and batch statistics are instead used during evaluation time as well.

Note: This `momentum` argument is different from one used in optimizer classes and the conventional notion of momentum. Mathematically, the update rule for running statistics here is $\hat{x}_{\text{new}} = (1 - \text{momentum}) \times \hat{x} + \text{momemtum} \times x_t$, where $\hat{x}$ is the estimated statistic and x_t is the new observed value.

Because the Batch Normalization is done over the *C* dimension, computing statistics on *(N, +)* slices, its common terminology to call this Volumetric Batch Normalization or Spatio-temporal Batch Normalization.

Currently SyncBatchNorm only supports DistributedDataParallel with single GPU per process. Use torch.nn.utils.convert_sync_batchnorm() to convert BatchNorm layer to SyncBatchNorm before wrapping Network with DDP.

Parameters

- **num_features** – C from an expected input of size $(N, C, +)$
- **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. Can be set to `None` for cumulative moving average (i.e. simple average). Default: 0.1
- **affine** – a boolean value that when set to `True`, this module has learnable affine parameters. Default: `True`
- **track_running_stats** – a boolean value that when set to `True`, this module tracks the running mean and variance, and when set to `False`, this module does not track such statistics and always uses batch statistics in both training and eval modes. Default: `True`
- **process_group** – synchronization of stats happen within each process group individually. Default behavior is synchronization across the whole world

Shape:

- Input: $(N, C, +)$
- Output: $(N, C, +)$ (same shape as input)

Examples:

```
>>> # With Learnable Parameters
>>> m = nn.SyncBatchNorm(100)
>>> # creating process group (optional)
>>> # process_ids is a list of int identifying rank ids.
>>> process_group = torch.distributed.new_group(process_ids)
>>> # Without Learnable Parameters
>>> m = nn.BatchNorm3d(100, affine=False, process_group=process_group)
>>> input = torch.randn(20, 100, 35, 45, 10)
>>> output = m(input)

>>> # network is nn.BatchNorm layer
>>> sync_bn_network = torch.nn.utils.convert_sync_batchnorm(network, process_
↪group)
>>> # only single gpu per process is currently supported
>>> ddp_sync_bn_network = torch.nn.parallel.DistributedDataParallel(
>>>                         sync_bn_network,
>>>                         device_ids=[args.local_rank],
>>>                         output_device=args.local_rank)
```

16.8.6 InstanceNorm1d

class torch.nn.**InstanceNorm1d**(*num_features*, *eps=1e-05*, *momentum=0.1*, *affine=False*, *track_running_stats=False*)

Applies Instance Normalization over a 3D input (a mini-batch of 1D inputs with optional additional channel dimension) as described in the paper Instance Normalization: The Missing Ingredient for Fast Stylization .

$$y = \frac{x - \mathrm{E}[x]}{\sqrt{\mathrm{Var}[x] + \epsilon}} * \gamma + \beta$$

The mean and standard-deviation are calculated per-dimension separately for each object in a mini-batch. γ and β are learnable parameter vectors of size *C* (where *C* is the input size) if `affine` is `True`.

By default, this layer uses instance statistics computed from input data in both training and evaluation modes.

If `track_running_stats` is set to `True`, during training this layer keeps running estimates of its computed mean and variance, which are then used for normalization during evaluation. The running estimates are kept with a default `momentum` of 0.1.

Note: This `momentum` argument is different from one used in optimizer classes and the conventional notion of momentum. Mathematically, the update rule for running statistics here is $\hat{x}_{\text{new}} = (1 - \text{momentum}) \times \hat{x} + \text{momemtum} \times x_t$, where $\hat{x}$ is the estimated statistic and x_t is the new observed value.

Note: *InstanceNorm1d* and *LayerNorm* are very similar, but have some subtle differences. *InstanceNorm1d* is applied on each channel of channeled data like multidimensional time series, but *LayerNorm* is usually applied on entire sample and often in NLP tasks. Additionaly, *LayerNorm* applies elementwise affine transform, while *InstanceNorm1d* usually dont apply affine transform.

Parameters

- **num_features** – C from an expected input of size (N, C, L) or L from input of size (N, L)
- **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`, this module has learnable affine parameters, initialized the same way as done for batch normalization. Default: `False`.
- **track_running_stats** – a boolean value that when set to `True`, this module tracks the running mean and variance, and when set to `False`, this module does not track such statistics and always uses batch statistics in both training and eval modes. Default: `False`

Shape:

- Input: (N, C, L)
- Output: (N, C, L) (same shape as input)

Examples:

```
>>> # Without Learnable Parameters
>>> m = nn.InstanceNorm1d(100)
>>> # With Learnable Parameters
>>> m = nn.InstanceNorm1d(100, affine=True)
>>> input = torch.randn(20, 100, 40)
>>> output = m(input)
```

16.8.7 InstanceNorm2d

class torch.nn.**InstanceNorm2d**(*num_features*, *eps=1e-05*, *momentum=0.1*, *affine=False*, *track_running_stats=False*)

Applies Instance Normalization over a 4D input (a mini-batch of 2D inputs with additional channel dimension) as described in the paper Instance Normalization: The Missing Ingredient for Fast Stylization .

$$y = \frac{x - \mathrm{E}[x]}{\sqrt{\mathrm{Var}[x] + \epsilon}} * \gamma + \beta$$

The mean and standard-deviation are calculated per-dimension separately for each object in a mini-batch. γ and β are learnable parameter vectors of size *C* (where *C* is the input size) if `affine` is `True`.

By default, this layer uses instance statistics computed from input data in both training and evaluation modes.

If `track_running_stats` is set to `True`, during training this layer keeps running estimates of its computed mean and variance, which are then used for normalization during evaluation. The running estimates are kept with a default `momentum` of 0.1.

Note: This `momentum` argument is different from one used in optimizer classes and the conventional notion of momentum. Mathematically, the update rule for running statistics here is $\hat{x}_{\text{new}} = (1 - \text{momentum}) \times \hat{x} + \text{momemtum} \times x_t$, where $\hat{x}$ is the estimated statistic and x_t is the new observed value.

Note: *InstanceNorm2d* and *LayerNorm* are very similar, but have some subtle differences. *InstanceNorm2d* is applied on each channel of channeled data like RGB images, but *LayerNorm* is usually applied on entire sample and often in NLP tasks. Additionaly, *LayerNorm* applies elementwise affine transform, while *InstanceNorm2d* usually dont apply affine transform.

Parameters

- **num_features** – C from an expected input of size (N, C, H, W)
- **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`, this module has learnable affine parameters, initialized the same way as done for batch normalization. Default: `False`.
- **track_running_stats** – a boolean value that when set to `True`, this module tracks the running mean and variance, and when set to `False`, this module does not track such statistics and always uses batch statistics in both training and eval modes. Default: `False`

Shape:

- Input: (N, C, H, W)
- Output: (N, C, H, W) (same shape as input)

Examples:

```
>>> # Without Learnable Parameters
>>> m = nn.InstanceNorm2d(100)
>>> # With Learnable Parameters
>>> m = nn.InstanceNorm2d(100, affine=True)
>>> input = torch.randn(20, 100, 35, 45)
>>> output = m(input)
```

16.8.8 InstanceNorm3d

class torch.nn.**InstanceNorm3d**(*num_features*, *eps=1e-05*, *momentum=0.1*, *affine=False*, *track_running_stats=False*)

Applies Instance Normalization over a 5D input (a mini-batch of 3D inputs with additional channel dimension) as described in the paper Instance Normalization: The Missing Ingredient for Fast Stylization .

$$y = \frac{x - \mathrm{E}[x]}{\sqrt{\mathrm{Var}[x] + \epsilon}} * \gamma + \beta$$

The mean and standard-deviation are calculated per-dimension separately for each object in a mini-batch. γ and β are learnable parameter vectors of size C (where C is the input size) if `affine` is `True`.

By default, this layer uses instance statistics computed from input data in both training and evaluation modes.

If `track_running_stats` is set to `True`, during training this layer keeps running estimates of its computed mean and variance, which are then used for normalization during evaluation. The running estimates are kept with a default `momentum` of 0.1.

Note: This `momentum` argument is different from one used in optimizer classes and the conventional notion of momentum. Mathematically, the update rule for running statistics here is $\hat{x}_{\text{new}} = (1 - \text{momentum}) \times \hat{x} + \text{momemtum} \times x_t$, where $\hat{x}$ is the estimated statistic and x_t is the new observed value.

Note: *InstanceNorm3d* and *LayerNorm* are very similar, but have some subtle differences. *InstanceNorm3d* is applied on each channel of channeled data like 3D models with RGB color, but *LayerNorm* is usually applied on entire sample and often in NLP tasks. Additionaly, *LayerNorm* applies elementwise affine transform, while *InstanceNorm3d* usually dont apply affine transform.

Parameters

- **num_features** – C from an expected input of size (N, C, D, H, W)
- **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`, this module has learnable affine parameters, initialized the same way as done for batch normalization. Default: `False`.
- **track_running_stats** – a boolean value that when set to `True`, this module tracks the running mean and variance, and when set to `False`, this module does not track such statistics and always uses batch statistics in both training and eval modes. Default: `False`

Shape:

- Input: (N, C, D, H, W)
- Output: (N, C, D, H, W) (same shape as input)

Examples:

```
>>> # Without Learnable Parameters
>>> m = nn.InstanceNorm3d(100)
>>> # With Learnable Parameters
>>> m = nn.InstanceNorm3d(100, affine=True)
>>> input = torch.randn(20, 100, 35, 45, 10)
>>> output = m(input)
```

16.8.9 LayerNorm

class `torch.nn.`**LayerNorm**(*normalized_shape*, *eps=1e-05*, *elementwise_affine=True*)

Applies Layer Normalization over a mini-batch of inputs as described in the paper Layer Normalization .

$$y = \frac{x - \mathrm{E}[x]}{\sqrt{\mathrm{Var}[x] + \epsilon}} * \gamma + \beta$$

The mean and standard-deviation are calculated separately over the last certain number dimensions which have to be of the shape specified by `normalized_shape`. γ and β are learnable affine transform parameters of `normalized_shape` if `elementwise_affine` is `True`.

Note: Unlike Batch Normalization and Instance Normalization, which applies scalar scale and bias for each entire channel/plane with the `affine` option, Layer Normalization applies per-element scale and bias with `elementwise_affine`.

This layer uses statistics computed from input data in both training and evaluation modes.

Parameters

- **normalized_shape** (*int or list or torch.Size*) – input shape from an expected input of size

$$[* \times \text{normalized\_shape}[0] \times \text{normalized\_shape}[1] \times \ldots \times \text{normalized\_shape}[-1]]$$

If a single integer is used, it is treated as a singleton list, and this module will normalize over the last dimension which is expected to be of that specific size.

- **eps** – a value added to the denominator for numerical stability. Default: 1e-5
- **elementwise_affine** – a boolean value that when set to `True`, this module has learnable per-element affine parameters initialized to ones (for weights) and zeros (for biases). Default: `True`.

Shape:

- Input: $(N, *)$
- Output: $(N, *)$ (same shape as input)

Examples:

```
>>> input = torch.randn(20, 5, 10, 10)
>>> # With Learnable Parameters
>>> m = nn.LayerNorm(input.size()[1:])
>>> # Without Learnable Parameters
>>> m = nn.LayerNorm(input.size()[1:], elementwise_affine=False)
>>> # Normalize over last two dimensions
>>> m = nn.LayerNorm([10, 10])
>>> # Normalize over last dimension of size 10
>>> m = nn.LayerNorm(10)
>>> # Activating the module
>>> output = m(input)
```

16.8.10 LocalResponseNorm

class torch.nn.**LocalResponseNorm**(*size*, *alpha=0.0001*, *beta=0.75*, *k=1.0*)

Applies local response normalization over an input signal composed of several input planes, where channels occupy the second dimension. Applies normalization across channels.

$$b_c = a_c \left(k + \frac{\alpha}{n} \sum_{c'=\max(0, c-n/2)}^{\min(N-1, c+n/2)} a_{c'}^2 \right)^{-\beta}$$

Parameters

- **size** – amount of neighbouring channels used for normalization
- **alpha** – multiplicative factor. Default: 0.0001
- **beta** – exponent. Default: 0.75
- **k** – additive factor. Default: 1

Shape:

- Input: $(N, C, *)$
- Output: $(N, C, *)$ (same shape as input)

Examples:

```
>>> lrn = nn.LocalResponseNorm(2)
>>> signal_2d = torch.randn(32, 5, 24, 24)
>>> signal_4d = torch.randn(16, 5, 7, 7, 7, 7)
>>> output_2d = lrn(signal_2d)
>>> output_4d = lrn(signal_4d)
```

16.9 Recurrent layers

16.9.1 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*, x_t is the input at time *t*, and $h_{(t-1)}$ is the hidden state of the previous layer at time *t-1* or the initial hidden state at time *0*. If `nonlinearity` is `'relu'`, then *ReLU* is used instead of *tanh*.

Parameters

- **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. E.g., setting `num_layers=2` would mean stacking two RNNs together to form a *stacked RNN*, with the second RNN taking in outputs of the first RNN and computing the final results. Default: 1
- **nonlinearity** – The non-linearity to use. Can be either `'tanh'` or `'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)*. Default: `False`
- **dropout** – If non-zero, introduces a *Dropout* layer on the outputs of each RNN layer except the last layer, with dropout probability equal to `dropout`. Default: 0
- **bidirectional** – If `True`, becomes a bidirectional RNN. Default: `False`

Inputs: input, h_0

- **input** of shape *(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()* or *torch.nn.utils.rnn.pack_sequence()* for details.
- **h_0** of shape *(num_layers * num_directions, batch, hidden_size)*: tensor containing the initial hidden state for each element in the batch. Defaults to zero if not provided. If the RNN is bidirectional, num_directions should be 2, else it should be 1.

Outputs: output, h_n

- **output** of shape *(seq_len, batch, num_directions * hidden_size)*: 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.

  For the unpacked case, the directions can be separated using `output.view(seq_len, batch, num_directions, hidden_size)`, with forward and backward being direction *0* and *1* respectively. Similarly, the directions can be separated in the packed case.

- **h_n** of shape *(num_layers * num_directions, batch, hidden_size)*: tensor containing the hidden state for *t = seq_len*.

Like *output*, the layers can be separated using `h_n.view(num_layers, num_directions, batch, hidden_size)`.

Shape:

- Input1: (L, N, H_{in}) tensor containing input features where H_{in} = input_size and L represents a sequence length.
- Input2: (S, N, H_{out}) tensor containing the initial hidden state for each element in the batch. H_{out} = hidden_size Defaults to zero if not provided. where S = num_layers $*$ num_directions If the RNN is bidirectional, num_directions should be 2, else it should be 1.
- Output1: (L, N, H_{all}) where H_all = num_directions $*$ hidden_size
- Output2: (S, N, H_{out}) tensor containing the next hidden state for each element in the batch

Variables

- **`weight_ih_l[k]`** – the learnable input-hidden weights of the k-th layer, of shape *(hidden_size, input_size)* for *k = 0*. Otherwise, the shape is *(hidden_size, num_directions * hidden_size)*
- **`weight_hh_l[k]`** – the learnable hidden-hidden weights of the k-th layer, of shape *(hidden_size, 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)*

Note: All the weights and biases are initialized from $\mathcal{U}(-\sqrt{k}, \sqrt{k})$ where $k = \frac{1}{\text{hidden\_size}}$

Note: If the following conditions are satisfied: 1) cudnn is enabled, 2) input data is on the GPU 3) input data has dtype `torch.float16` 4) V100 GPU is used, 5) input data is not in `PackedSequence` format persistent algorithm can be selected to improve performance.

Examples:

```
>>> rnn = nn.RNN(10, 20, 2)
>>> input = torch.randn(5, 3, 10)
>>> h0 = torch.randn(2, 3, 20)
>>> output, hn = rnn(input, h0)
```

16.9.2 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:

$$
\begin{aligned}
i_t &= \sigma(W_{ii}x_t + b_{ii} + W_{hi}h_{(t-1)} + b_{hi}) \\
f_t &= \sigma(W_{if}x_t + b_{if} + W_{hf}h_{(t-1)} + b_{hf}) \\
g_t &= \tanh(W_{ig}x_t + b_{ig} + W_{hg}h_{(t-1)} + b_{hg}) \\
o_t &= \sigma(W_{io}x_t + b_{io} + W_{ho}h_{(t-1)} + b_{ho}) \\
c_t &= f_t * c_{(t-1)} + i_t * g_t \\
h_t &= o_t * \tanh(c_t)
\end{aligned}
$$

where h_t is the hidden state at time *t*, c_t is the cell state at time *t*, x_t is the input at time *t*, $h_{(t-1)}$ is the hidden state of the layer at time *t-1* or the initial hidden state at time *0*, and i_t, f_t, g_t, o_t are the input, forget, cell, and output gates, respectively. σ is the sigmoid function, and $*$ is the Hadamard product.

In a multilayer LSTM, the input $x_t^{(l)}$ of the l -th layer ($l >= 2$) is the hidden state $h_t^{(l-1)}$ of the previous layer multiplied by dropout $\delta_t^{(l-1)}$ where each $\delta_t^{(l-1)}$ is a Bernoulli random variable which is 0 with probability `dropout`.

Parameters

- **`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. E.g., setting `num_layers=2` would mean stacking two LSTMs together to form a *stacked LSTM*, with the second LSTM taking in outputs of the first LSTM and computing the final results. Default: 1
- **`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). Default: `False`
- **`dropout`** – If non-zero, introduces a *Dropout* layer on the outputs of each LSTM layer except the last layer, with dropout probability equal to `dropout`. Default: 0
- **`bidirectional`** – If `True`, becomes a bidirectional LSTM. Default: `False`

Inputs: input, (h_0, c_0)

- **input** of shape *(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()`* or *`torch.nn.utils.rnn.pack_sequence()`* for details.
- **h_0** of shape *(num_layers * num_directions, batch, hidden_size)*: tensor containing the initial hidden state for each element in the batch. If the LSTM is bidirectional, num_directions should be 2, else it should be 1.
- **c_0** of shape *(num_layers * num_directions, batch, hidden_size)*: tensor containing the initial cell state for each element in the batch.

  If *(h_0, c_0)* is not provided, both **h_0** and **c_0** default to zero.

Outputs: output, (h_n, c_n)

- **output** of shape *(seq_len, batch, num_directions * hidden_size)*: tensor containing the output features *(h_t)* from the last layer of the LSTM, for each *t*. If a *`torch.nn.utils.rnn.PackedSequence`* has been given as the input, the output will also be a packed sequence.

  For the unpacked case, the directions can be separated using `output.view(seq_len, batch, num_directions, hidden_size)`, with forward and backward being direction *0* and *1* respectively. Similarly, the directions can be separated in the packed case.

- **h_n** of shape *(num_layers * num_directions, batch, hidden_size)*: tensor containing the hidden state for *t = seq_len*.

  Like *output*, the layers can be separated using `h_n.view(num_layers, num_directions, batch, hidden_size)` and similarly for *c_n*.

- **c_n** of shape *(num_layers * num_directions, batch, hidden_size)*: tensor containing the cell state for *t = seq_len*.

Variables

- **weight_ih_l[k]** – the learnable input-hidden weights of the k^{th} layer *(W_ii|W_if|W_ig|W_io)*, of shape *(4*hidden_size, input_size)* for *k = 0*. Otherwise, the shape is *(4*hidden_size, num_directions * 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 *(4*hidden_size, 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 *(b_hi|b_hf|b_hg|b_ho)*, of shape *(4*hidden_size)*

Note: All the weights and biases are initialized from $\mathcal{U}(-\sqrt{k}, \sqrt{k})$ where $k = \frac{1}{\text{hidden\_size}}$

Note: If the following conditions are satisfied: 1) cudnn is enabled, 2) input data is on the GPU 3) input data has dtype `torch.float16` 4) V100 GPU is used, 5) input data is not in `PackedSequence` format persistent algorithm can be selected to improve performance.

Examples:

```
>>> rnn = nn.LSTM(10, 20, 2)
>>> input = torch.randn(5, 3, 10)
>>> h0 = torch.randn(2, 3, 20)
>>> c0 = torch.randn(2, 3, 20)
>>> output, (hn, cn) = rnn(input, (h0, c0))
```

16.9.3 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:

$$
\begin{aligned}
r_t &= \sigma(W_{ir} x_t + b_{ir} + W_{hr} h_{(t-1)} + b_{hr}) \\
z_t &= \sigma(W_{iz} x_t + b_{iz} + W_{hz} h_{(t-1)} + b_{hz}) \\
n_t &= \tanh(W_{in} x_t + b_{in} + r_t * (W_{hn} h_{(t-1)} + b_{hn})) \\
h_t &= (1 - z_t) * n_t + z_t * h_{(t-1)}
\end{aligned}
$$

where h_t is the hidden state at time *t*, x_t is the input at time *t*, $h_{(t-1)}$ is the hidden state of the layer at time *t-1* or the initial hidden state at time *0*, and r_t, z_t, n_t are the reset, update, and new gates, respectively. σ is the sigmoid function, and $*$ is the Hadamard product.

In a multilayer GRU, the input $x_t^{(l)}$ of the l -th layer ($l >= 2$) is the hidden state $h_t^{(l-1)}$ of the previous layer multiplied by dropout $\delta_t^{(l-1)}$ where each $\delta_t^{(l-1)}$ is a Bernoulli random variable which is 0 with probability `dropout`.

Parameters

- **`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. E.g., setting `num_layers=2` would mean stacking two GRUs together to form a *stacked GRU*, with the second GRU taking in outputs of the first GRU and computing the final results. Default: 1
- **`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). Default: `False`
- **`dropout`** – If non-zero, introduces a *Dropout* layer on the outputs of each GRU layer except the last layer, with dropout probability equal to `dropout`. Default: 0
- **`bidirectional`** – If `True`, becomes a bidirectional GRU. Default: `False`

Inputs: input, h_0

- **input** of shape *(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** of shape *(num_layers * num_directions, batch, hidden_size)*: tensor containing the initial hidden state for each element in the batch. Defaults to zero if not provided. If the RNN is bidirectional, num_directions should be 2, else it should be 1.

Outputs: output, h_n

- **output** of shape *(seq_len, batch, num_directions * hidden_size)*: tensor containing the output features h_t from the last layer of the GRU, for each *t*. If a *`torch.nn.utils.rnn.PackedSequence`* has been given as the input, the output will also be a packed sequence. For the unpacked case, the directions can be separated using `output.view(seq_len, batch, num_directions, hidden_size)`, with forward and backward being direction *0* and *1* respectively.

  Similarly, the directions can be separated in the packed case.
- **h_n** of shape *(num_layers * num_directions, batch, hidden_size)*: tensor containing the hidden state for *t = seq_len*

  Like *output*, the layers can be separated using `h_n.view(num_layers, num_directions, batch, hidden_size)`.

Shape:

- Input1: (L, N, H_{in}) tensor containing input features where H_{in} = input_size and *L* represents a sequence length.
- Input2: (S, N, H_{out}) tensor containing the initial hidden state for each element in the batch. H_{out} = hidden_size Defaults to zero if not provided. where S = num_layers $*$ num_directions If the RNN is bidirectional, num_directions should be 2, else it should be 1.
- Output1: (L, N, H_{all}) where H_all = num_directions $*$ hidden_size
- Output2: (S, N, H_{out}) tensor containing the next hidden state for each element in the batch

Variables

- **weight_ih_l[k]** – the learnable input-hidden weights of the kth layer (W_ir|W_iz|W_in), of shape *(3*hidden_size, input_size)* for *k = 0*. Otherwise, the shape is *(3*hidden_size, num_directions * hidden_size)*
- **weight_hh_l[k]** – the learnable hidden-hidden weights of the kth layer (W_hr|W_hz|W_hn), of shape *(3*hidden_size, hidden_size)*
- **bias_ih_l[k]** – the learnable input-hidden bias of the kth layer (b_ir|b_iz|b_in), of shape *(3*hidden_size)*
- **bias_hh_l[k]** – the learnable hidden-hidden bias of the kth layer (b_hr|b_hz|b_hn), of shape *(3*hidden_size)*

Note: All the weights and biases are initialized from $\mathcal{U}(-\sqrt{k}, \sqrt{k})$ where $k = \frac{1}{\text{hidden\_size}}$

Note: If the following conditions are satisfied: 1) cudnn is enabled, 2) input data is on the GPU 3) input data has dtype `torch.float16` 4) V100 GPU is used, 5) input data is not in `PackedSequence` format persistent algorithm can be selected to improve performance.

Examples:

```
>>> rnn = nn.GRU(10, 20, 2)
>>> input = torch.randn(5, 3, 10)
>>> h0 = torch.randn(2, 3, 20)
>>> output, hn = rnn(input, h0)
```

16.9.4 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` is *relu*, then ReLU is used in place of tanh.

Parameters

- **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. Can be either `'tanh'` or `'relu'`. Default: `'tanh'`

Inputs: input, hidden

- **input** of shape *(batch, input_size)*: tensor containing input features
- **hidden** of shape *(batch, hidden_size)*: tensor containing the initial hidden state for each element in the batch. Defaults to zero if not provided.

Outputs: h

- **h** of shape *(batch, hidden_size)*: tensor containing the next hidden state for each element in the batch

Shape:

- Input1: (N, H_{in}) tensor containing input features where H_{in} = *input_size*
- Input2: (N, H_{out}) tensor containing the initial hidden state for each element in the batch where H_{out} = *hidden_size* Defaults to zero if not provided.
- Output: (N, H_{out}) tensor containing the next hidden state for each element in the batch

Variables

- **weight_ih** – the learnable input-hidden weights, of shape *(hidden_size, input_size)*
- **weight_hh** – the learnable hidden-hidden weights, of shape *(hidden_size, hidden_size)*
- **bias_ih** – the learnable input-hidden bias, of shape *(hidden_size)*
- **bias_hh** – the learnable hidden-hidden bias, of shape *(hidden_size)*

Note: All the weights and biases are initialized from $\mathcal{U}(-\sqrt{k}, \sqrt{k})$ where $k = \frac{1}{\text{hidden\_size}}$

Examples:

```
>>> rnn = nn.RNNCell(10, 20)
>>> input = torch.randn(6, 3, 10)
>>> hx = torch.randn(3, 20)
>>> output = []
>>> for i in range(6):
        hx = rnn(input[i], hx)
        output.append(hx)
```

16.9.5 LSTMCell

class torch.nn.**LSTMCell**(*input_size*, *hidden_size*, *bias=True*)

A long short-term memory (LSTM) cell.

$$
\begin{aligned}
i &= \sigma(W_{ii}x + b_{ii} + W_{hi}h + b_{hi}) \\
f &= \sigma(W_{if}x + b_{if} + W_{hf}h + b_{hf}) \\
g &= \tanh(W_{ig}x + b_{ig} + W_{hg}h + b_{hg}) \\
o &= \sigma(W_{io}x + b_{io} + W_{ho}h + b_{ho}) \\
c' &= f * c + i * g \\
h' &= o * \tanh(c')
\end{aligned}
$$

where σ is the sigmoid function, and $*$ is the Hadamard product.

Parameters

- **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** of shape *(batch, input_size)*: tensor containing input features
- **h_0** of shape *(batch, hidden_size)*: tensor containing the initial hidden state for each element in the batch.

- **c_0** of shape *(batch, hidden_size)*: tensor containing the initial cell state for each element in the batch.

  If *(h_0, c_0)* is not provided, both **h_0** and **c_0** default to zero.

Outputs: (h_1, c_1)

- **h_1** of shape *(batch, hidden_size)*: tensor containing the next hidden state for each element in the batch
- **c_1** of shape *(batch, hidden_size)*: tensor containing the next cell state for each element in the batch

Variables

- **weight_ih** – the learnable input-hidden weights, of shape *(4*hidden_size, input_size)*
- **weight_hh** – the learnable hidden-hidden weights, of shape *(4*hidden_size, hidden_size)*
- **bias_ih** – the learnable input-hidden bias, of shape *(4*hidden_size)*
- **bias_hh** – the learnable hidden-hidden bias, of shape *(4*hidden_size)*

Note: All the weights and biases are initialized from $\mathcal{U}(-\sqrt{k}, \sqrt{k})$ where $k = \frac{1}{\text{hidden\_size}}$

Examples:

```
>>> rnn = nn.LSTMCell(10, 20)
>>> input = torch.randn(6, 3, 10)
>>> hx = torch.randn(3, 20)
>>> cx = torch.randn(3, 20)
>>> output = []
>>> for i in range(6):
        hx, cx = rnn(input[i], (hx, cx))
        output.append(hx)
```

16.9.6 GRUCell

class torch.nn.**GRUCell**(*input_size*, *hidden_size*, *bias=True*)

A gated recurrent unit (GRU) cell

$$
\begin{aligned}
r &= \sigma(W_{ir}x + b_{ir} + W_{hr}h + b_{hr}) \\
z &= \sigma(W_{iz}x + b_{iz} + W_{hz}h + b_{hz}) \\
n &= \tanh(W_{in}x + b_{in} + r * (W_{hn}h + b_{hn})) \\
h' &= (1 - z) * n + z * h
\end{aligned}
$$

where σ is the sigmoid function, and $*$ is the Hadamard product.

Parameters

- **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** of shape *(batch, input_size)*: tensor containing input features
- **hidden** of shape *(batch, hidden_size)*: tensor containing the initial hidden state for each element in the batch. Defaults to zero if not provided.

Outputs: h

- **h** of shape *(batch, hidden_size)*: tensor containing the next hidden state for each element in the batch

Shape:

- Input1: (N, H_{in}) tensor containing input features where H_{in} = *input_size*
- Input2: (N, H_{out}) tensor containing the initial hidden state for each element in the batch where H_{out} = *hidden_size* Defaults to zero if not provided.
- Output: (N, H_{out}) tensor containing the next hidden state for each element in the batch

Variables

- **weight_ih** – the learnable input-hidden weights, of shape *(3*hidden_size, input_size)*
- **weight_hh** – the learnable hidden-hidden weights, of shape *(3*hidden_size, hidden_size)*
- **bias_ih** – the learnable input-hidden bias, of shape *(3*hidden_size)*
- **bias_hh** – the learnable hidden-hidden bias, of shape *(3*hidden_size)*

Note: All the weights and biases are initialized from $\mathcal{U}(-\sqrt{k}, \sqrt{k})$ where $k = \frac{1}{\text{hidden\_size}}$

Examples:

```
>>> rnn = nn.GRUCell(10, 20)
>>> input = torch.randn(6, 3, 10)
>>> hx = torch.randn(3, 20)
>>> output = []
>>> for i in range(6):
        hx = rnn(input[i], hx)
        output.append(hx)
```

16.10 Transformer layers

16.10.1 Transformer

16.10.2 TransformerEncoder

16.10.3 TransformerDecoder

16.10.4 TransformerEncoderLayer

16.10.5 TransformerDecoderLayer

16.11 Linear layers

16.11.1 Identity

16.11.2 Linear

class torch.nn.**Linear**(*in_features*, *out_features*, *bias=True*)

Applies a linear transformation to the incoming data: $y = xA^T + b$

Parameters

- **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: $(N, *, H_{in})$ where $*$ means any number of additional dimensions and $H_{in} =$ in_features
- Output: $(N, *, H_{out})$ where all but the last dimension are the same shape as the input and $H_{out} =$ out_features.

Variables

- **weight** – the learnable weights of the module of shape $(\text{out\_features}, \text{in\_features})$. The values are initialized from $\mathcal{U}(-\sqrt{k}, \sqrt{k})$, where $k = \frac{1}{\text{in\_features}}$
- **bias** – the learnable bias of the module of shape (out_features). If `bias` is `True`, the values are initialized from $\mathcal{U}(-\sqrt{k}, \sqrt{k})$ where $k = \frac{1}{\text{in\_features}}$

Examples:

```
>>> m = nn.Linear(20, 30)
>>> input = torch.randn(128, 20)
>>> output = m(input)
>>> print(output.size())
torch.Size([128, 30])
```

16.11.3 Bilinear

class torch.nn.**Bilinear**(*in1_features*, *in2_features*, *out_features*, *bias=True*)

Applies a bilinear transformation to the incoming data: $y = x_1 A x_2 + b$

Parameters

- **in1_features** – size of each first input sample
- **in2_features** – size of each second 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:

- Input1: $(N, *, H_{in1})$ where $H_{in1} =$ in1_features and $*$ means any number of additional dimensions. All but the last dimension of the inputs should be the same.
- Input2: $(N, *, H_{in2})$ where $H_{in2} =$ in2_features.
- Output: $(N, *, H_{out})$ where $H_{out} =$ out_features and all but the last dimension are the same shape as the input.

Variables

- **weight** – the learnable weights of the module of shape $(\text{out\_features}, \text{in1\_features}, \text{in2\_features})$. The values are initialized from $\mathcal{U}(-\sqrt{k}, \sqrt{k})$, where $k = \frac{1}{\text{in1\_features}}$
- **bias** – the learnable bias of the module of shape (out_features). If `bias` is `True`, the values are initialized from $\mathcal{U}(-\sqrt{k}, \sqrt{k})$, where $k = \frac{1}{\text{in1\_features}}$

Examples:

```
>>> m = nn.Bilinear(20, 30, 40)
>>> input1 = torch.randn(128, 20)
>>> input2 = torch.randn(128, 30)
>>> output = m(input1, input2)
>>> print(output.size())
torch.Size([128, 40])
```

16.12 Dropout layers

16.12.1 Dropout

class torch.nn.**Dropout**(*p=0.5*, *inplace=False*)

During training, randomly zeroes some of the elements of the input tensor with probability `p` using samples from a Bernoulli distribution. Each channel will be zeroed out independently on every forward call.

This has proven to be an effective technique for regularization and preventing the co-adaptation of neurons as described in the paper Improving neural networks by preventing co-adaptation of feature detectors .

Furthermore, the outputs are scaled by a factor of $\frac{1}{1-p}$ during training. This means that during evaluation the module simply computes an identity function.

Parameters

- **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: $(*)$. Input can be of any shape
- Output: $(*)$. Output is of the same shape as input

Examples:

```
>>> m = nn.Dropout(p=0.2)
>>> input = torch.randn(20, 16)
>>> output = m(input)
```

16.12.2 Dropout2d

class `torch.nn.`**Dropout2d**(*p=0.5*, *inplace=False*)

Randomly zero out entire channels (a channel is a 2D feature map, e.g., the j-th channel of the i-th sample in the batched input is a 2D tensor input$[i, j]$). Each channel will be zeroed out independently on every forward call with probability `p` using samples from a Bernoulli distribution.

Usually the input comes from `nn.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 i.i.d. 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.

Parameters

- **p** (*float, optional*) – probability of an element to be zero-ed.
- **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 = torch.randn(20, 16, 32, 32)
>>> output = m(input)
```

16.12.3 Dropout3d

class `torch.nn.`**Dropout3d**(*p=0.5*, *inplace=False*)

Randomly zero out entire channels (a channel is a 3D feature map, e.g., the j-th channel of the i-th sample in the batched input is a 3D tensor input$[i, j]$). Each channel will be zeroed out independently on every forward call with probability `p` using samples from a Bernoulli distribution.

Usually the input comes from `nn.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 i.i.d. 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.

Parameters

- **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 = torch.randn(20, 16, 4, 32, 32)
>>> output = m(input)
```

16.12.4 AlphaDropout

class `torch.nn.`**AlphaDropout**(*p=0.5*, *inplace=False*)

Applies Alpha Dropout over the input.

Alpha Dropout is a type of Dropout that maintains the self-normalizing property. For an input with zero mean and unit standard deviation, the output of Alpha Dropout maintains the original mean and standard deviation of the input. Alpha Dropout goes hand-in-hand with SELU activation function, which ensures that the outputs have zero mean and unit standard deviation.

During training, it randomly masks some of the elements of the input tensor with probability *p* using samples from a bernoulli distribution. The elements to masked are randomized on every forward call, and scaled and shifted to maintain zero mean and unit standard deviation.

During evaluation the module simply computes an identity function.

More details can be found in the paper Self-Normalizing Neural Networks .

Parameters

- **p** (*float*) – probability of an element to be dropped. Default: 0.5
- **inplace** (*bool, optional*) – If set to `True`, will do this operation in-place

Shape:

- Input: $(*)$. Input can be of any shape
- Output: $(*)$. Output is of the same shape as input

Examples:

```
>>> m = nn.AlphaDropout(p=0.2)
>>> input = torch.randn(20, 16)
>>> output = m(input)
```

16.13 Sparse layers

16.13.1 Embedding

class torch.nn.**Embedding**(*num_embeddings*, *embedding_dim*, *padding_idx=None*, *max_norm=None*, *norm_type=2.0*, *scale_grad_by_freq=False*, *sparse=False*, *_weight=None*)

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.

Parameters

- **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 the embedding vector at `padding_idx` (initialized to zeros) whenever it encounters the index.
- **max_norm** (*float, optional*) – If given, each embedding vector with norm larger than `max_norm` is renormalized to have norm `max_norm`.
- **norm_type** (*float, optional*) – The p of the p-norm to compute for the `max_norm` option. Default `2`.
- **scale_grad_by_freq** (*boolean, optional*) – If given, this will scale gradients by the inverse of frequency of the words in the mini-batch. Default `False`.
- **sparse** (*bool, optional*) – If `True`, gradient w.r.t. `weight` matrix will be a sparse tensor. See Notes for more details regarding sparse gradients.

Variables **weight** (`Tensor`) – the learnable weights of the module of shape (num_embeddings, embedding_dim) initialized from $\mathcal{N}(0, 1)$

Shape:

- Input: $(*)$, LongTensor of arbitrary shape containing the indices to extract
- Output: $(*, H)$, where * is the input shape and $H =$ embedding_dim

---

Note: Keep in mind that only a limited number of optimizers support sparse gradients: currently its `optim.SGD` (*CUDA* and *CPU*), `optim.SparseAdam` (*CUDA* and *CPU*) and `optim.Adagrad` (*CPU*)

Note: With `padding_idx` set, the embedding vector at `padding_idx` is initialized to all zeros. However, note that this vector can be modified afterwards, e.g., using a customized initialization method, and thus changing the vector used to pad the output. The gradient for this vector from *Embedding* is always zero.

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 = torch.LongTensor([[1,2,4,5],[4,3,2,9]])
>>> embedding(input)
```

(continues on next page)

(continued from previous page)

```
tensor([[[-0.0251, -1.6902,  0.7172],
         [-0.6431,  0.0748,  0.6969],
         [ 1.4970,  1.3448, -0.9685],
         [-0.3677, -2.7265, -0.1685]],

        [[ 1.4970,  1.3448, -0.9685],
         [ 0.4362, -0.4004,  0.9400],
         [-0.6431,  0.0748,  0.6969],
         [ 0.9124, -2.3616,  1.1151]]])


>>> # example with padding_idx
>>> embedding = nn.Embedding(10, 3, padding_idx=0)
>>> input = torch.LongTensor([[0,2,0,5]])
>>> embedding(input)
tensor([[[ 0.0000,  0.0000,  0.0000],
         [ 0.1535, -2.0309,  0.9315],
         [ 0.0000,  0.0000,  0.0000],
         [-0.1655,  0.9897,  0.0635]]])
```

classmethod from_pretrained(*embeddings*, *freeze=True*, *padding_idx=None*, *max_norm=None*, *norm_type=2.0*, *scale_grad_by_freq=False*, *sparse=False*)

Creates Embedding instance from given 2-dimensional FloatTensor.

Parameters

- **embeddings** (Tensor) – FloatTensor containing weights for the Embedding. First dimension is being passed to Embedding as `num_embeddings`, second as `embedding_dim`.
- **freeze** (*boolean, optional*) – If `True`, the tensor does not get updated in the learning process. Equivalent to `embedding.weight.requires_grad = False`. Default: `True`
- **padding_idx** (*int, optional*) – See module initialization documentation.
- **max_norm** (*float, optional*) – See module initialization documentation.
- **norm_type** (*float, optional*) – See module initialization documentation. Default `2`.
- **scale_grad_by_freq** (*boolean, optional*) – See module initialization documentation. Default `False`.
- **sparse** (*bool, optional*) – See module initialization documentation.

Examples:

```
>>> # FloatTensor containing pretrained weights
>>> weight = torch.FloatTensor([[1, 2.3, 3], [4, 5.1, 6.3]])
>>> embedding = nn.Embedding.from_pretrained(weight)
>>> # Get embeddings for index 1
>>> input = torch.LongTensor([1])
>>> embedding(input)
tensor([[ 4.0000,  5.1000,  6.3000]])
```

16.13.2 EmbeddingBag

class torch.nn.**EmbeddingBag**(*num_embeddings*, *embedding_dim*, *max_norm=None*, *norm_type=2.0*, *scale_grad_by_freq=False*, *mode='mean'*, *sparse=False*, *_weight=None*)

Computes sums or means of bags of embeddings, without instantiating the intermediate embeddings.

For bags of constant length, this class

- with `mode="sum"` is equivalent to *Embedding* followed by `torch.sum(dim=0)`,
- with `mode="mean"` is equivalent to *Embedding* followed by `torch.mean(dim=0)`,
- with `mode="max"` is equivalent to *Embedding* followed by `torch.max(dim=0)`.

However, *EmbeddingBag* is much more time and memory efficient than using a chain of these operations.

Parameters

- **num_embeddings** (*int*) – size of the dictionary of embeddings
- **embedding_dim** (*int*) – the size of each embedding vector
- **max_norm** (*float, optional*) – If given, each embedding vector with norm larger than `max_norm` is renormalized to have norm `max_norm`.
- **norm_type** (*float, optional*) – The p of the p-norm to compute for the `max_norm` option. Default `2`.
- **scale_grad_by_freq** (*boolean, optional*) – if given, this will scale gradients by the inverse of frequency of the words in the mini-batch. Default `False`. Note: this option is not supported when `mode="max"`.
- **mode** (*string, optional*) – `"sum"`, `"mean"` or `"max"`. Specifies the way to reduce the bag. Default: `"mean"`
- **sparse** (*bool, optional*) – if `True`, gradient w.r.t. `weight` matrix will be a sparse tensor. See Notes for more details regarding sparse gradients. Note: this option is not supported when `mode="max"`.

Variables **weight** (`Tensor`) – the learnable weights of the module of shape *(num_embeddings, embedding_dim)* initialized from $\mathcal{N}(0, 1)$.

Inputs: `input` (LongTensor) and `offsets` (LongTensor, optional)

- If `input` is 2D of shape *(B, N)*,

  it will be treated as `B` bags (sequences) each of fixed length `N`, and this will return `B` values aggregated in a way depending on the `mode`. `offsets` is ignored and required to be `None` in this case.

- If `input` is 1D of shape *(N)*,

  it will be treated as a concatenation of multiple bags (sequences). `offsets` is required to be a 1D tensor containing the starting index positions of each bag in `input`. Therefore, for `offsets` of shape *(B)*, `input` will be viewed as having `B` bags. Empty bags (i.e., having 0-length) will have returned vectors filled by zeros.

Output shape: *(B, embedding_dim)*

Examples:

```
>>> # an Embedding module containing 10 tensors of size 3
>>> embedding_sum = nn.EmbeddingBag(10, 3, mode='sum')
>>> # a batch of 2 samples of 4 indices each
```

(continues on next page)

(continued from previous page)

```
>>> input = torch.LongTensor([1,2,4,5,4,3,2,9])
>>> offsets = torch.LongTensor([0,4])
>>> embedding_sum(input, offsets)
tensor([[-0.8861, -5.4350, -0.0523],
        [ 1.1306, -2.5798, -1.0044]])
```

classmethod from_pretrained(*embeddings*, *freeze=True*, *max_norm=None*, *norm_type=2.0*, *scale_grad_by_freq=False*, *mode='mean'*, *sparse=False*)

Creates EmbeddingBag instance from given 2-dimensional FloatTensor.

Parameters

- **embeddings** (Tensor) – FloatTensor containing weights for the EmbeddingBag. First dimension is being passed to EmbeddingBag as num_embeddings, second as embedding_dim.
- **freeze** (*boolean, optional*) – If `True`, the tensor does not get updated in the learning process. Equivalent to `embeddingbag.weight.requires_grad = False`. Default: `True`
- **max_norm** (*float, optional*) – See module initialization documentation. Default: `None`
- **norm_type** (*float, optional*) – See module initialization documentation. Default `2`.
- **scale_grad_by_freq** (*boolean, optional*) – See module initialization documentation. Default `False`.
- **mode** (*string, optional*) – See module initialization documentation. Default: `"mean"`
- **sparse** (*bool, optional*) – See module initialization documentation. Default: `False`.

Examples:

```
>>> # FloatTensor containing pretrained weights
>>> weight = torch.FloatTensor([[1, 2.3, 3], [4, 5.1, 6.3]])
>>> embeddingbag = nn.EmbeddingBag.from_pretrained(weight)
>>> # Get embeddings for index 1
>>> input = torch.LongTensor([[1, 0]])
>>> embeddingbag(input)
tensor([[ 2.5000,  3.7000,  4.6500]])
```

16.14 Distance functions

16.14.1 CosineSimilarity

class torch.nn.**CosineSimilarity**(*dim=1*, *eps=1e-08*)

Returns cosine similarity between x_1 and x_2, computed along dim.

$$\text{similarity} = \frac{x_1 \cdot x_2}{\max(\|x_1\|_2 \cdot \|x_2\|_2, \epsilon)}$$

Parameters

- **dim** (*int, optional*) – Dimension where cosine similarity is computed. Default: 1

- **eps** (*float, optional*) – Small value to avoid division by zero. Default: 1e-8

Shape:

- Input1: $(*_1, D, *_2)$ where D is at position *dim*
- Input2: $(*_1, D, *_2)$, same shape as the Input1
- Output: $(*_1, *_2)$

Examples:

```
>>> input1 = torch.randn(100, 128)
>>> input2 = torch.randn(100, 128)
>>> cos = nn.CosineSimilarity(dim=1, eps=1e-6)
>>> output = cos(input1, input2)
```

16.14.2 PairwiseDistance

class torch.nn.**PairwiseDistance**(*p=2.0*, *eps=1e-06*, *keepdim=False*)

Computes the batchwise pairwise distance between vectors v_1, v_2 using the p-norm:

$$\|x\|_p = \left(\sum_{i=1}^{n} |x_i|^p \right)^{1/p}$$

Parameters

- **p** (*real*) – the norm degree. Default: 2
- **eps** (*float, optional*) – Small value to avoid division by zero. Default: 1e-6
- **keepdim** (*bool, optional*) – Determines whether or not to keep the batch dimension. Default: False

Shape:

- Input1: (N, D) where *D = vector dimension*
- Input2: (N, D), same shape as the Input1
- Output: (N). If `keepdim` is `False`, then $(N, 1)$.

Examples:

```
>>> pdist = nn.PairwiseDistance(p=2)
>>> input1 = torch.randn(100, 128)
>>> input2 = torch.randn(100, 128)
>>> output = pdist(input1, input2)
```

16.15 Loss functions

16.15.1 L1Loss

class torch.nn.**L1Loss**(*size_average=None*, *reduce=None*, *reduction='mean'*)

Creates a criterion that measures the mean absolute error (MAE) between each element in the input x and target y.

The unreduced (i.e. with `reduction` set to `'none'`) loss can be described as:

$$\ell(x, y) = L = \{l_1, \dots, l_N\}^\top, \quad l_n = |x_n - y_n|,$$

where N is the batch size. If `reduction` is not `'none'` (default `'mean'`), then:

$$\ell(x, y) = \begin{cases} \text{mean}(L), & \text{if reduction} = \text{'mean'}; \\ \text{sum}(L), & \text{if reduction} = \text{'sum'}. \end{cases}$$

x and y are tensors of 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 `reduction = 'sum'`.

Parameters

- **size_average** (*bool, optional*) – Deprecated (see `reduction`). By default, the losses are averaged over each loss element in the batch. Note that for some losses, there are multiple elements per sample. If the field `size_average` is set to `False`, the losses are instead summed for each minibatch. Ignored when reduce is `False`. Default: `True`
- **reduce** (*bool, optional*) – Deprecated (see `reduction`). By default, the losses are averaged or summed over observations for each minibatch depending on `size_average`. When `reduce` is `False`, returns a loss per batch element instead and ignores `size_average`. Default: `True`
- **reduction** (*string, optional*) – Specifies the reduction to apply to the output: `'none'` | `'mean'` | `'sum'`. `'none'`: no reduction will be applied, `'mean'`: the sum of the output will be divided by the number of elements in the output, `'sum'`: the output will be summed. Note: `size_average` and `reduce` are in the process of being deprecated, and in the meantime, specifying either of those two args will override `reduction`. Default: `'mean'`

Shape:

- Input: $(N, *)$ where $*$ means, any number of additional dimensions
- Target: $(N, *)$, same shape as the input
- Output: scalar. If `reduction` is `'none'`, then $(N, *)$, same shape as the input

Examples:

```
>>> loss = nn.L1Loss()
>>> input = torch.randn(3, 5, requires_grad=True)
>>> target = torch.randn(3, 5)
>>> output = loss(input, target)
>>> output.backward()
```

16.15.2 MSELoss

class torch.nn.**MSELoss**(*size_average=None*, *reduce=None*, *reduction='mean'*)

Creates a criterion that measures the mean squared error (squared L2 norm) between each element in the input x and target y.

The unreduced (i.e. with `reduction` set to `'none'`) loss can be described as:

$$\ell(x, y) = L = \{l_1, \dots, l_N\}^\top, \quad l_n = (x_n - y_n)^2,$$

where N is the batch size. If `reduction` is not `'none'` (default `'mean'`), then:

$$\ell(x, y) = \begin{cases} \text{mean}(L), & \text{if reduction} = \text{'mean'}; \\ \text{sum}(L), & \text{if reduction} = \text{'sum'}. \end{cases}$$

x and y are tensors of 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 `reduction = 'sum'`.

Parameters

- **size_average** (*bool, optional*) – Deprecated (see `reduction`). By default, the losses are averaged over each loss element in the batch. Note that for some losses, there are multiple elements per sample. If the field `size_average` is set to `False`, the losses are instead summed for each minibatch. Ignored when reduce is `False`. Default: `True`
- **reduce** (*bool, optional*) – Deprecated (see `reduction`). By default, the losses are averaged or summed over observations for each minibatch depending on `size_average`. When `reduce` is `False`, returns a loss per batch element instead and ignores `size_average`. Default: `True`
- **reduction** (*string, optional*) – Specifies the reduction to apply to the output: `'none'` | `'mean'` | `'sum'`. `'none'`: no reduction will be applied, `'mean'`: the sum of the output will be divided by the number of elements in the output, `'sum'`: the output will be summed. Note: `size_average` and `reduce` are in the process of being deprecated, and in the meantime, specifying either of those two args will override `reduction`. Default: `'mean'`

Shape:

- Input: $(N, *)$ where $*$ means, any number of additional dimensions
- Target: $(N, *)$, same shape as the input

Examples:

```
>>> loss = nn.MSELoss()
>>> input = torch.randn(3, 5, requires_grad=True)
>>> target = torch.randn(3, 5)
>>> output = loss(input, target)
>>> output.backward()
```

16.15.3 CrossEntropyLoss

class torch.nn.**CrossEntropyLoss**(*weight=None*, *size_average=None*, *ignore_index=-100*, *reduce=None*, *reduction='mean'*)

This criterion combines `nn.LogSoftmax()` and `nn.NLLLoss()` in one single class.

It is useful when training a classification problem with *C* classes. If provided, the optional argument `weight` 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 raw, unnormalized scores for each class.

input has to be a Tensor of size either $(minibatch, C)$ or $(minibatch, C, d_1, d_2, ..., d_K)$ with $K \geq 1$ for the *K*-dimensional case (described later).

This criterion expects a class index in the range $[0, C-1]$ as the *target* for each value of a 1D tensor of size *minibatch*; if *ignore_index* is specified, this criterion also accepts this class index (this index may not necessarily be in the class range).

The loss can be described as:

$$\text{loss}(x, class) = -\log\left(\frac{\exp(x[class])}{\sum_j \exp(x[j])}\right) = -x[class] + \log\left(\sum_j \exp(x[j])\right)$$

or in the case of the `weight` argument being specified:

$$\text{loss}(x, class) = weight[class]\left(-x[class] + \log\left(\sum_j \exp(x[j])\right)\right)$$

The losses are averaged across observations for each minibatch.

Can also be used for higher dimension inputs, such as 2D images, by providing an input of size $(minibatch, C, d_1, d_2, ..., d_K)$ with $K \geq 1$, where K is the number of dimensions, and a target of appropriate shape (see below).

Parameters

- **weight** (*Tensor, optional*) – a manual rescaling weight given to each class. If given, has to be a Tensor of size *C*
- **size_average** (*bool, optional*) – Deprecated (see `reduction`). By default, the losses are averaged over each loss element in the batch. Note that for some losses, there are multiple elements per sample. If the field `size_average` is set to `False`, the losses are instead summed for each minibatch. Ignored when reduce is `False`. Default: `True`
- **ignore_index** (*int, optional*) – Specifies a target value that is ignored and does not contribute to the input gradient. When `size_average` is `True`, the loss is averaged over non-ignored targets.
- **reduce** (*bool, optional*) – Deprecated (see `reduction`). By default, the losses are averaged or summed over observations for each minibatch depending on `size_average`. When `reduce` is `False`, returns a loss per batch element instead and ignores `size_average`. Default: `True`
- **reduction** (*string, optional*) – Specifies the reduction to apply to the output: `'none' | 'mean' | 'sum'`. `'none'`: no reduction will be applied, `'mean'`: the sum of the output will be divided by the number of elements in the output, `'sum'`: the output will be summed. Note: `size_average` and `reduce` are in the process of being deprecated, and in the meantime, specifying either of those two args will override `reduction`. Default: `'mean'`

Shape:

- Input: (N, C) where *C = number of classes*, or $(N, C, d_1, d_2, ..., d_K)$ with $K \geq 1$ in the case of *K*-dimensional loss.
- Target: (N) where each value is $0 \leq \text{targets}[i] \leq C-1$, or $(N, d_1, d_2, ..., d_K)$ with $K \geq 1$ in the case of K-dimensional loss.
- Output: scalar. If `reduction` is `'none'`, then the same size as the target: (N), or $(N, d_1, d_2, ..., d_K)$ with $K \geq 1$ in the case of K-dimensional loss.

Examples:

```
>>> loss = nn.CrossEntropyLoss()
>>> input = torch.randn(3, 5, requires_grad=True)
>>> target = torch.empty(3, dtype=torch.long).random_(5)
>>> output = loss(input, target)
>>> output.backward()
```

16.15.4 CTCLoss

class torch.nn.**CTCLoss**(*blank=0*, *reduction='mean'*, *zero_infinity=False*)

The Connectionist Temporal Classification loss.

Calculates loss between a continuous (unsegmented) time series and a target sequence. CTCLoss sums over the probability of possible alignments of input to target, producing a loss value which is differentiable with respect to each input node. The alignment of input to target is assumed to be many-to-one, which limits the length of the target sequence such that it must be :math: *leq* the input length.

Args: **blank** (int, optional): blank label. Default 0. reduction (string, optional): Specifies the reduction to apply to the output:

`'none'|'mean'|'sum'`. `'none'`: no reduction will be applied, `'mean'`: the output losses will be divided by the target lengths and then the mean over the batch is taken. Default: `'mean'`

zero_infinity (bool, optional): Whether to zero infinite losses and the associated gradients. Default: `False` Infinite losses mainly occur when the inputs are too short to be aligned to the targets.

Inputs:

log_probs: Tensor of size (T, N, C)

$T = inputlength$
$N = batchsize$
$C = numberofclasses(includingblank)$

The logarithmized probabilities of the outputs (e.g. obtained with *torch.nn.functional.log_softmax()*).

targets: Tensor of size (N, S) **or** ***(sum(target_lengths))***

$N = batchsize$
$S = maxtargetlength, if shapeis(N, S).$

Target sequences. Each element in the target sequence is a class index. Target index cannot be blank (default=0).

In the (N, S) form, targets are padded to the length of the longest sequence, and stacked.
In the $(sum(targetlengths))$ form, the targets are assumed to be un-padded and concatenated within 1 dimension.

input_lengths: Tuple or tensor of size (N)**.** Lengths of the inputs (must each be $\leq T$). Lengths are specified for each sequence to achieve masking under the assumption that sequences are padded to equal lengths.

target_lengths: Tuple or tensor of size (N)**.**

Lengths of the targets. Lengths are specified for each sequence to achieve masking under the assumption that sequences are padded to equal lengths.

If target shape is (N, S), target_lengths are effectively the stop index s_n for each target sequence, such that `target_n = targets[n,0:s_n]` for each target in a batch. Lengths must each be $\leq S$

If the targets are given as a 1d tensor that is the concatenation of individual targets, the target_lengths must add up to the total length of the tensor.

Example:

```
>>> T = 50      # Input sequence length
>>> C = 20      # Number of classes (excluding blank)
>>> N = 16      # Batch size
>>> S = 30      # Target sequence length of longest target in batch
>>> S_min = 10  # Minimum target length, for demonstration purposes
>>>
>>> # Initialize random batch of input vectors, for *size = (T,N,C)
>>> input = torch.randn(T, N, C).log_softmax(2).detach().requires_grad_()
>>>
>>> # Initialize random batch of targets (0 = blank, 1:C+1 = classes)
>>> target = torch.randint(low=1, high=C+1, size=(N, S), dtype=torch.long)
>>>
>>> input_lengths = torch.full(size=(N,), fill_value=T, dtype=torch.long)
>>> target_lengths = torch.randint(low=S_min, high=S, size=(N,), dtype=torch.long)
>>> ctc_loss = nn.CTCLoss()
>>> loss = ctc_loss(input, target, input_lengths, target_lengths)
>>> loss.backward()
```

Reference: A. Graves et al.: Connectionist Temporal Classification: Labelling Unsegmented Sequence Data with Recurrent Neural Networks: https://www.cs.toronto.edu/~graves/icml_2006.pdf

---

Note: In order to use CuDNN, the following must be satisfied: `targets` must be in concatenated format, all `input_lengths` must be *T*. *blank* $= 0$, `target_lengths` ≤ 256, the integer arguments must be of dtype `torch.int32`.

The regular implementation uses the (more common in PyTorch) *torch.long* dtype.

Note: In some circumstances when using the CUDA backend with CuDNN, this operator may select a nondeterministic algorithm to increase performance. If this is undesirable, you can try to make the operation deterministic (potentially at a performance cost) by setting `torch.backends.cudnn.deterministic = True`. Please see the notes on *Reproducibility* for background.

16.15.5 NLLLoss

class torch.nn.**NLLLoss**(*weight=None*, *size_average=None*, *ignore_index=-100*, *reduce=None*, *reduction='mean'*)

The negative log likelihood loss. It is useful to train a classification problem with *C* classes.

If provided, the optional argument `weight` 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 Tensor of size either $(minibatch, C)$ or $(minibatch, C, d_1, d_2, ..., d_K)$ with $K \geq 1$ for the *K*-dimensional case (described later).

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 should be a class index in the range $[0, C-1]$ where *C = number of classes*; if *ignore_index* is specified, this loss also accepts this class index (this index may not necessarily be in the class range).

The unreduced (i.e. with `reduction` set to `'none'`) loss can be described as:

$$\ell(x, y) = L = \{l_1, \dots, l_N\}^\top, \quad l_n = -w_{y_n} x_{n,y_n}, \quad w_c = \text{weight}[c] \cdot \mathbb{1}\{c \neq \text{ignore\_index}\},$$

where N is the batch size. If `reduction` is not `'none'` (default `'mean'`), then

$$\ell(x, y) = \begin{cases} \sum_{n=1}^{N} \frac{1}{\sum_{n=1}^{N} w_{y_n}} l_n, & \text{if reduction} = \text{'mean'}; \\ \sum_{n=1}^{N} l_n, & \text{if reduction} = \text{'sum'}. \end{cases}$$

Can also be used for higher dimension inputs, such as 2D images, by providing an input of size $(minibatch, C, d_1, d_2, ..., d_K)$ with $K \geq 1$, where K is the number of dimensions, and a target of appropriate shape (see below). In the case of images, it computes NLL loss per-pixel.

Parameters

- **weight** (*Tensor, optional*) – a manual rescaling weight given to each class. If given, it has to be a Tensor of size *C*. Otherwise, it is treated as if having all ones.
- **size_average** (*bool, optional*) – Deprecated (see `reduction`). By default, the losses are averaged over each loss element in the batch. Note that for some losses, there are multiple elements per sample. If the field `size_average` is set to `False`, the losses are instead summed for each minibatch. Ignored when reduce is `False`. Default: `True`
- **ignore_index** (*int, optional*) – Specifies a target value that is ignored and does not contribute to the input gradient. When `size_average` is `True`, the loss is averaged over non-ignored targets.
- **reduce** (*bool, optional*) – Deprecated (see `reduction`). By default, the losses are averaged or summed over observations for each minibatch depending on `size_average`. When `reduce` is `False`, returns a loss per batch element instead and ignores `size_average`. Default: `True`
- **reduction** (*string, optional*) – Specifies the reduction to apply to the output: `'none'` | `'mean'` | `'sum'`. `'none'`: no reduction will be applied, `'mean'`: the sum of the output will be divided by the number of elements in the output, `'sum'`: the output will be summed. Note: `size_average` and `reduce` are in the process of being deprecated, and in the meantime, specifying either of those two args will override `reduction`. Default: `'mean'`

Shape:

- Input: (N, C) where *C = number of classes*, or $(N, C, d_1, d_2, ..., d_K)$ with $K \geq 1$ in the case of *K*-dimensional loss.
- Target: (N) where each value is $0 \leq \text{targets}[i] \leq C-1$, or $(N, d_1, d_2, ..., d_K)$ with $K \geq 1$ in the case of K-dimensional loss.

- Output: scalar. If `reduction` is `'none'`, then the same size as the target: (N), or $(N, d_1, d_2, ..., d_K)$ with $K \geq 1$ in the case of K-dimensional loss.

Examples:

```
>>> m = nn.LogSoftmax(dim=1)
>>> loss = nn.NLLLoss()
>>> # input is of size N x C = 3 x 5
>>> input = torch.randn(3, 5, requires_grad=True)
>>> # each element in target has to have 0 <= value < C
>>> target = torch.tensor([1, 0, 4])
>>> output = loss(m(input), target)
>>> output.backward()
>>>
>>>
>>> # 2D loss example (used, for example, with image inputs)
>>> N, C = 5, 4
>>> loss = nn.NLLLoss()
>>> # input is of size N x C x height x width
>>> data = torch.randn(N, 16, 10, 10)
>>> conv = nn.Conv2d(16, C, (3, 3))
>>> m = nn.LogSoftmax(dim=1)
>>> # each element in target has to have 0 <= value < C
>>> target = torch.empty(N, 8, 8, dtype=torch.long).random_(0, C)
>>> output = loss(m(conv(data)), target)
>>> output.backward()
```

16.15.6 PoissonNLLLoss

class torch.nn.**PoissonNLLLoss**(*log_input=True*, *full=False*, *size_average=None*, *eps=1e-08*, *reduce=None*, *reduction='mean'*)

Negative log likelihood loss with Poisson distribution of target.

The loss can be described as:

$$\text{target} \sim \text{Poisson}(\text{input})$$
$$\text{loss}(\text{input}, \text{target}) = \text{input} - \text{target} * \log(\text{input}) + \log(\text{target!})$$

The last term can be omitted or approximated with Stirling formula. The approximation is used for target values more than 1. For targets less or equal to 1 zeros are added to the loss.

Parameters

- **log_input** (*bool, optional*) – if `True` the loss is computed as $\exp(\text{input}) - \text{target} * \text{input}$, if `False` the loss is $\text{input} - \text{target} * \log(\text{input} + \text{eps})$.
- **full** (*bool, optional*) – whether to compute full loss, i. e. to add the Stirling approximation term

$$\text{target} * \log(\text{target}) - \text{target} + 0.5 * \log(2\pi\text{target}).$$

- **size_average** (*bool, optional*) – Deprecated (see `reduction`). By default, the losses are averaged over each loss element in the batch. Note that for some losses, there are multiple elements per sample. If the field `size_average` is set to `False`, the losses are instead summed for each minibatch. Ignored when reduce is `False`. Default: `True`
- **eps** (*float, optional*) – Small value to avoid evaluation of $\log(0)$ when `log_input = False`. Default: 1e-8

- **reduce** (*bool, optional*) – Deprecated (see `reduction`). By default, the losses are averaged or summed over observations for each minibatch depending on `size_average`. When `reduce` is `False`, returns a loss per batch element instead and ignores `size_average`. Default: `True`
- **reduction** (*string, optional*) – Specifies the reduction to apply to the output: `'none'` | `'mean'` | `'sum'`. `'none'`: no reduction will be applied, `'mean'`: the sum of the output will be divided by the number of elements in the output, `'sum'`: the output will be summed. Note: `size_average` and `reduce` are in the process of being deprecated, and in the meantime, specifying either of those two args will override `reduction`. Default: `'mean'`

Examples:

```
>>> loss = nn.PoissonNLLLoss()
>>> log_input = torch.randn(5, 2, requires_grad=True)
>>> target = torch.randn(5, 2)
>>> output = loss(log_input, target)
>>> output.backward()
```

Shape:

- Input: $(N, *)$ where $*$ means, any number of additional dimensions
- Target: $(N, *)$, same shape as the input
- Output: scalar by default. If `reduction` is `'none'`, then $(N, *)$, the same shape as the input

16.15.7 KLDivLoss

class torch.nn.**KLDivLoss**(*size_average=None*, *reduce=None*, *reduction='mean'*)

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* and is not restricted to a 2D Tensor. The targets are given as *probabilities* (i.e. without taking the logarithm).

This criterion expects a *target Tensor* of the same size as the *input Tensor*.

The unreduced (i.e. with `reduction` set to `'none'`) loss can be described as:

$$l(x, y) = L = \{l_1, \dots, l_N\}, \quad l_n = y_n \cdot (\log y_n - x_n)$$

where the index N spans all dimensions of `input` and L has the same shape as `input`. If `reduction` is not `'none'` (default `'mean'`), then:

$$\ell(x, y) = \begin{cases} \text{mean}(L), & \text{if reduction} = \text{'mean'}; \\ \text{sum}(L), & \text{if reduction} = \text{'sum'}. \end{cases}$$

In default `reduction` mode `'mean'`, the losses are averaged for each minibatch over observations **as well as** over dimensions. `'batchmean'` mode gives the correct KL divergence where losses are averaged over batch dimension only. `'mean'` modes behavior will be changed to the same as `'batchmean'` in the next major release.

Parameters

- **size_average** (*bool, optional*) – Deprecated (see `reduction`). By default, the losses are averaged over each loss element in the batch. Note that for some losses, there are multiple elements per sample. If the field `size_average` is set to `False`, the losses are instead summed for each minibatch. Ignored when reduce is `False`. Default: `True`
- **reduce** (*bool, optional*) – Deprecated (see `reduction`). By default, the losses are averaged or summed over observations for each minibatch depending on `size_average`. When `reduce` is `False`, returns a loss per batch element instead and ignores `size_average`. Default: `True`
- **reduction** (*string, optional*) – Specifies the reduction to apply to the output: `'none'` | `'batchmean'` | `'sum'` | `'mean'`. `'none'`: no reduction will be applied. `'batchmean'`: the sum of the output will be divided by batchsize. `'sum'`: the output will be summed. `'mean'`: the output will be divided by the number of elements in the output. Default: `'mean'`

Note: `size_average` and `reduce` are in the process of being deprecated, and in the meantime, specifying either of those two args will override `reduction`.

Note: `reduction = 'mean'` doesnt return the true kl divergence value, please use `reduction = 'batchmean'` which aligns with KL math definition. In the next major release, `'mean'` will be changed to be the same as `'batchmean'`.

Shape:

- Input: $(N, *)$ where $*$ means, any number of additional dimensions
- Target: $(N, *)$, same shape as the input
- Output: scalar by default. If :attr:`reduction` is `'none'`, then $(N, *)$, the same shape as the input

16.15.8 BCELoss

class torch.nn.**BCELoss**(*weight=None*, *size_average=None*, *reduce=None*, *reduction='mean'*)

Creates a criterion that measures the Binary Cross Entropy between the target and the output:

The unreduced (i.e. with `reduction` set to `'none'`) loss can be described as:

$$\ell(x, y) = L = \{l_1, \dots, l_N\}^\top, \quad l_n = -w_n \left[y_n \cdot \log x_n + (1 - y_n) \cdot \log(1 - x_n) \right],$$

where N is the batch size. If `reduction` is not `'none'` (default `'mean'`), then

$$\ell(x, y) = \begin{cases} \text{mean}(L), & \text{if reduction} = \text{'mean'}; \\ \text{sum}(L), & \text{if reduction} = \text{'sum'}. \end{cases}$$

This is used for measuring the error of a reconstruction in for example an auto-encoder. Note that the targets y should be numbers between 0 and 1.

Parameters

- **weight** (Tensor, *optional*) – a manual rescaling weight given to the loss of each batch element. If given, has to be a Tensor of size *nbatch*.

- **size_average** (*bool, optional*) – Deprecated (see `reduction`). By default, the losses are averaged over each loss element in the batch. Note that for some losses, there are multiple elements per sample. If the field `size_average` is set to `False`, the losses are instead summed for each minibatch. Ignored when reduce is `False`. Default: `True`
- **reduce** (*bool, optional*) – Deprecated (see `reduction`). By default, the losses are averaged or summed over observations for each minibatch depending on `size_average`. When `reduce` is `False`, returns a loss per batch element instead and ignores `size_average`. Default: `True`
- **reduction** (*string, optional*) – Specifies the reduction to apply to the output: `'none' | 'mean' | 'sum'`. `'none'`: no reduction will be applied, `'mean'`: the sum of the output will be divided by the number of elements in the output, `'sum'`: the output will be summed. Note: `size_average` and `reduce` are in the process of being deprecated, and in the meantime, specifying either of those two args will override `reduction`. Default: `'mean'`

Shape:

- Input: $(N, *)$ where $*$ means, any number of additional dimensions
- Target: $(N, *)$, same shape as the input
- Output: scalar. If `reduction` is `'none'`, then $(N, *)$, same shape as input.

Examples:

```
>>> m = nn.Sigmoid()
>>> loss = nn.BCELoss()
>>> input = torch.randn(3, requires_grad=True)
>>> target = torch.empty(3).random_(2)
>>> output = loss(m(input), target)
>>> output.backward()
```

16.15.9 BCEWithLogitsLoss

class torch.nn.**BCEWithLogitsLoss**(*weight=None, size_average=None, reduce=None, reduction='mean', pos_weight=None*)

This loss combines a *Sigmoid* layer and the *BCELoss* in one single class. This version is more numerically stable than using a plain *Sigmoid* followed by a *BCELoss* as, by combining the operations into one layer, we take advantage of the log-sum-exp trick for numerical stability.

The unreduced (i.e. with `reduction` set to `'none'`) loss can be described as:

$$\ell(x, y) = L = \{l_1, \dots, l_N\}^\top, \quad l_n = -w_n \left[y_n \cdot \log \sigma(x_n) + (1 - y_n) \cdot \log(1 - \sigma(x_n)) \right],$$

where N is the batch size. If `reduction` is not `'none'` (default `'mean'`), then

$$\ell(x, y) = \begin{cases} \text{mean}(L), & \text{if reduction} = \text{'mean'}; \\ \text{sum}(L), & \text{if reduction} = \text{'sum'}. \end{cases}$$

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.

Its possible to trade off recall and precision by adding weights to positive examples. In this case the loss can be described as:

$$\ell(x, y) = L = \{l_1, \dots, l_N\}^\top, \quad l_n = -w_n \left[p_n y_n \cdot \log \sigma(x_n) + (1 - y_n) \cdot \log(1 - \sigma(x_n)) \right],$$

where p_n is the weight of the positive class for sample n in the batch. $p_n > 1$ increases the recall, $p_n < 1$ increases the precision.

For example, if a dataset contains 100 positive and 300 negative examples of a single class, then *pos_weight* for the class should be equal to $\frac{300}{100} = 3$. The loss would act as if the dataset contains $3 \times 100 = 300$ positive examples.

Parameters

- **weight** (Tensor, *optional*) – a manual rescaling weight given to the loss of each batch element. If given, has to be a Tensor of size *nbatch*.
- **size_average** (*bool, optional*) – Deprecated (see `reduction`). By default, the losses are averaged over each loss element in the batch. Note that for some losses, there are multiple elements per sample. If the field `size_average` is set to `False`, the losses are instead summed for each minibatch. Ignored when reduce is `False`. Default: `True`
- **reduce** (*bool, optional*) – Deprecated (see `reduction`). By default, the losses are averaged or summed over observations for each minibatch depending on `size_average`. When `reduce` is `False`, returns a loss per batch element instead and ignores `size_average`. Default: `True`
- **reduction** (*string, optional*) – Specifies the reduction to apply to the output: `'none'` | `'mean'` | `'sum'`. `'none'`: no reduction will be applied, `'mean'`: the sum of the output will be divided by the number of elements in the output, `'sum'`: the output will be summed. Note: `size_average` and `reduce` are in the process of being deprecated, and in the meantime, specifying either of those two args will override `reduction`. Default: `'mean'`
- **pos_weight** (Tensor, *optional*) – a weight of positive examples. Must be a vector with length equal to the number of classes.

Shape:

- Input: $(N, *)$ where $*$ means, any number of additional dimensions
- Target: $(N, *)$, same shape as the input
- Output: scalar. If `reduction` is `'none'`, then $(N, *)$, same shape as input.

Examples:

```
>>> loss = nn.BCEWithLogitsLoss()
>>> input = torch.randn(3, requires_grad=True)
>>> target = torch.empty(3).random_(2)
>>> output = loss(input, target)
>>> output.backward()
```

16.15.10 MarginRankingLoss

class torch.nn.**MarginRankingLoss**(*margin=0.0, size_average=None, reduce=None, reduction='mean'*)

Creates a criterion that measures the loss given inputs $x1$, $x2$, two 1D mini-batch *Tensors*, and a label 1D mini-batch tensor y (containing 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:

$$\text{loss}(x, y) = \max(0, -y * (x1 - x2) + \text{margin})$$

Parameters

- **margin** (*float, optional*) – Has a default value of 0.
- **size_average** (*bool, optional*) – Deprecated (see `reduction`). By default, the losses are averaged over each loss element in the batch. Note that for some losses, there are multiple elements per sample. If the field `size_average` is set to `False`, the losses are instead summed for each minibatch. Ignored when reduce is `False`. Default: `True`
- **reduce** (*bool, optional*) – Deprecated (see `reduction`). By default, the losses are averaged or summed over observations for each minibatch depending on `size_average`. When `reduce` is `False`, returns a loss per batch element instead and ignores `size_average`. Default: `True`
- **reduction** (*string, optional*) – Specifies the reduction to apply to the output: `'none'` | `'mean'` | `'sum'`. `'none'`: no reduction will be applied, `'mean'`: the sum of the output will be divided by the number of elements in the output, `'sum'`: the output will be summed. Note: `size_average` and `reduce` are in the process of being deprecated, and in the meantime, specifying either of those two args will override `reduction`. Default: `'mean'`

Shape:

- Input: (N, D) where *N* is the batch size and *D* is the size of a sample.
- Target: (N)
- Output: scalar. If `reduction` is `'none'`, then (N).

16.15.11 HingeEmbeddingLoss

class torch.nn.**HingeEmbeddingLoss**(*margin=1.0*, *size_average=None*, *reduce=None*, *reduction='mean'*)

Measures the loss given an input tensor x and a labels tensor y (containing 1 or -1). This is usually used for measuring whether two inputs are similar or dissimilar, e.g. using the L1 pairwise distance as x, and is typically used for learning nonlinear embeddings or semi-supervised learning.

The loss function for n-th sample in the mini-batch is

$$l_n = \begin{cases} x_n, & \text{if } y_n = 1, \\ \max\{0, \Delta - x_n\}, & \text{if } y_n = -1, \end{cases}$$

and the total loss functions is

$$\ell(x, y) = \begin{cases} \text{mean}(L), & \text{if reduction} = \text{'mean'}; \\ \text{sum}(L), & \text{if reduction} = \text{'sum'}. \end{cases}$$

where $L = \{l_1, \ldots, l_N\}^\top$.

Parameters

- **margin** (*float, optional*) – Has a default value of *1*.
- **size_average** (*bool, optional*) – Deprecated (see `reduction`). By default, the losses are averaged over each loss element in the batch. Note that for some losses, there are multiple elements per sample. If the field `size_average` is set to `False`, the losses are instead summed for each minibatch. Ignored when reduce is `False`. Default: `True`

- **reduce** (*bool, optional*) – Deprecated (see `reduction`). By default, the losses are averaged or summed over observations for each minibatch depending on `size_average`. When `reduce` is `False`, returns a loss per batch element instead and ignores `size_average`. Default: `True`
- **reduction** (*string, optional*) – Specifies the reduction to apply to the output: `'none'` | `'mean'` | `'sum'`. `'none'`: no reduction will be applied, `'mean'`: the sum of the output will be divided by the number of elements in the output, `'sum'`: the output will be summed. Note: `size_average` and `reduce` are in the process of being deprecated, and in the meantime, specifying either of those two args will override `reduction`. Default: `'mean'`

Shape:

- Input: $(*)$ where $*$ means, any number of dimensions. The sum operation operates over all the elements.
- Target: $(*)$, same shape as the input
- Output: scalar. If `reduction` is `'none'`, then same shape as the input

16.15.12 MultiLabelMarginLoss

class torch.nn.**MultiLabelMarginLoss**(*size_average=None*, *reduce=None*, *reduction='mean'*)

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:

$$\text{loss}(x, y) = \sum_{ij} \frac{\max(0, 1 - (x[y[j]] - x[i]))}{\text{x.size}(0)}$$

where $x \in \{0, \cdots, \text{x.size}(0) - 1\}$, $y \in \{0, \cdots, \text{y.size}(0) - 1\}$, $0 \leq y[j] \leq \text{x.size}(0) - 1$, and $i \neq y[j]$ for all i and j.

y and x must have the same size.

The criterion only considers a contiguous block of non-negative targets that starts at the front.

This allows for different samples to have variable amounts of target classes.

Parameters

- **size_average** (*bool, optional*) – Deprecated (see `reduction`). By default, the losses are averaged over each loss element in the batch. Note that for some losses, there are multiple elements per sample. If the field `size_average` is set to `False`, the losses are instead summed for each minibatch. Ignored when reduce is `False`. Default: `True`
- **reduce** (*bool, optional*) – Deprecated (see `reduction`). By default, the losses are averaged or summed over observations for each minibatch depending on `size_average`. When `reduce` is `False`, returns a loss per batch element instead and ignores `size_average`. Default: `True`
- **reduction** (*string, optional*) – Specifies the reduction to apply to the output: `'none'` | `'mean'` | `'sum'`. `'none'`: no reduction will be applied, `'mean'`: the sum of the output will be divided by the number of elements in the output, `'sum'`: the output will be summed. Note: `size_average` and `reduce` are in the process of being deprecated, and in the meantime, specifying either of those two args will override `reduction`. Default: `'mean'`

Shape:

- Input: (C) or (N, C) where *N* is the batch size and *C* is the number of classes.
- Target: (C) or (N, C), label targets padded by -1 ensuring same shape as the input.
- Output: scalar. If `reduction` is `'none'`, then (N).

Examples:

```
>>> loss = nn.MultiLabelMarginLoss()
>>> x = torch.FloatTensor([[0.1, 0.2, 0.4, 0.8]])
>>> # for target y, only consider labels 3 and 0, not after label -1
>>> y = torch.LongTensor([[3, 0, -1, 1]])
>>> loss(x, y)
>>> # 0.25 * ((1-(0.1-0.2)) + (1-(0.1-0.4)) + (1-(0.8-0.2)) + (1-(0.8-0.4)))
tensor(0.8500)
```

16.15.13 SmoothL1Loss

class `torch.nn.`**SmoothL1Loss**(*size_average=None*, *reduce=None*, *reduction='mean'*)

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:

$$\text{loss}(x, y) = \frac{1}{n} \sum_i z_i$$

where z_i is given by:

$$z_i = \begin{cases} 0.5(x_i - y_i)^2, & \text{if } |x_i - y_i| < 1 \\ |x_i - y_i| - 0.5, & \text{otherwise} \end{cases}$$

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 sets `reduction = 'sum'`.

Parameters

- **size_average** (*bool, optional*) – Deprecated (see `reduction`). By default, the losses are averaged over each loss element in the batch. Note that for some losses, there are multiple elements per sample. If the field `size_average` is set to `False`, the losses are instead summed for each minibatch. Ignored when reduce is `False`. Default: `True`
- **reduce** (*bool, optional*) – Deprecated (see `reduction`). By default, the losses are averaged or summed over observations for each minibatch depending on `size_average`. When `reduce` is `False`, returns a loss per batch element instead and ignores `size_average`. Default: `True`
- **reduction** (*string, optional*) – Specifies the reduction to apply to the output: `'none'` | `'mean'` | `'sum'`. `'none'`: no reduction will be applied, `'mean'`: the sum of the output will be divided by the number of elements in the output, `'sum'`: the output will be summed. Note: `size_average` and `reduce` are in the process of being deprecated, and in the meantime, specifying either of those two args will override `reduction`. Default: `'mean'`

Shape:

- Input: $(N, *)$ where $*$ means, any number of additional dimensions
- Target: $(N, *)$, same shape as the input
- Output: scalar. If `reduction` is `'none'`, then $(N, *)$, same shape as the input

16.15.14 SoftMarginLoss

class torch.nn.**SoftMarginLoss**(*size_average=None*, *reduce=None*, *reduction='mean'*)

Creates a criterion that optimizes a two-class classification logistic loss between input tensor x and target tensor y (containing 1 or -1).

$$\text{loss}(x, y) = \sum_i \frac{\log(1 + \exp(-y[i] * x[i]))}{\text{x.nelement}()}$$

Parameters

- **size_average** (*bool, optional*) – Deprecated (see `reduction`). By default, the losses are averaged over each loss element in the batch. Note that for some losses, there are multiple elements per sample. If the field `size_average` is set to `False`, the losses are instead summed for each minibatch. Ignored when reduce is `False`. Default: `True`
- **reduce** (*bool, optional*) – Deprecated (see `reduction`). By default, the losses are averaged or summed over observations for each minibatch depending on `size_average`. When `reduce` is `False`, returns a loss per batch element instead and ignores `size_average`. Default: `True`
- **reduction** (*string, optional*) – Specifies the reduction to apply to the output: `'none'` | `'mean'` | `'sum'`. `'none'`: no reduction will be applied, `'mean'`: the sum of the output will be divided by the number of elements in the output, `'sum'`: the output will be summed. Note: `size_average` and `reduce` are in the process of being deprecated, and in the meantime, specifying either of those two args will override `reduction`. Default: `'mean'`

Shape:

- Input: $(*)$ where $*$ means, any number of additional dimensions
- Target: $(*)$, same shape as the input
- Output: scalar. If `reduction` is `'none'`, then same shape as the input

16.15.15 MultiLabelSoftMarginLoss

class torch.nn.**MultiLabelSoftMarginLoss**(*weight=None*, *size_average=None*, *reduce=None*, *reduction='mean'*)

Creates a criterion that optimizes a multi-label one-versus-all loss based on max-entropy, between input x and target y of size (N, C). For each sample in the minibatch:

$$loss(x, y) = -\frac{1}{C} * \sum_i y[i] * \log((1 + \exp(-x[i]))^{-1}) + (1 - y[i]) * \log\left(\frac{\exp(-x[i])}{(1 + \exp(-x[i]))}\right)$$

where $i \in \{0, \cdots, \text{x.nElement}() - 1\}$, $y[i] \in \{0, 1\}$.

Parameters

- **weight** (Tensor, *optional*) – a manual rescaling weight given to each class. If given, it has to be a Tensor of size *C*. Otherwise, it is treated as if having all ones.

- **size_average** (*bool, optional*) – Deprecated (see `reduction`). By default, the losses are averaged over each loss element in the batch. Note that for some losses, there are multiple elements per sample. If the field `size_average` is set to `False`, the losses are instead summed for each minibatch. Ignored when reduce is `False`. Default: `True`
- **reduce** (*bool, optional*) – Deprecated (see `reduction`). By default, the losses are averaged or summed over observations for each minibatch depending on `size_average`. When `reduce` is `False`, returns a loss per batch element instead and ignores `size_average`. Default: `True`
- **reduction** (*string, optional*) – Specifies the reduction to apply to the output: `'none'` | `'mean'` | `'sum'`. `'none'`: no reduction will be applied, `'mean'`: the sum of the output will be divided by the number of elements in the output, `'sum'`: the output will be summed. Note: `size_average` and `reduce` are in the process of being deprecated, and in the meantime, specifying either of those two args will override `reduction`. Default: `'mean'`

Shape:

- Input: (N, C) where *N* is the batch size and *C* is the number of classes.
- Target: (N, C), label targets padded by -1 ensuring same shape as the input.
- Output: scalar. If `reduction` is `'none'`, then (N).

16.15.16 CosineEmbeddingLoss

class torch.nn.**CosineEmbeddingLoss**(*margin=0.0*, *size_average=None*, *reduce=None*, *reduction='mean'*)

Creates a criterion that measures the loss given input tensors x_1, x_2 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.

The loss function for each sample is:

$$\text{loss}(x, y) = \begin{cases} 1 - \cos(x_1, x_2), & \text{if } y = 1 \\ \max(0, \cos(x_1, x_2) - \text{margin}), & \text{if } y = -1 \end{cases}$$

Parameters

- **margin** (*float, optional*) – Should be a number from -1 to 1, 0 to 0.5 is suggested. If `margin` is missing, the default value is 0.
- **size_average** (*bool, optional*) – Deprecated (see `reduction`). By default, the losses are averaged over each loss element in the batch. Note that for some losses, there are multiple elements per sample. If the field `size_average` is set to `False`, the losses are instead summed for each minibatch. Ignored when reduce is `False`. Default: `True`
- **reduce** (*bool, optional*) – Deprecated (see `reduction`). By default, the losses are averaged or summed over observations for each minibatch depending on `size_average`. When `reduce` is `False`, returns a loss per batch element instead and ignores `size_average`. Default: `True`
- **reduction** (*string, optional*) – Specifies the reduction to apply to the output: `'none'` | `'mean'` | `'sum'`. `'none'`: no reduction will be applied, `'mean'`: the sum of the output will be divided by the number of elements in the output, `'sum'`: the output will be summed. Note: `size_average` and `reduce` are in the process of being deprecated, and in the meantime, specifying either of those two args will override `reduction`. Default: `'mean'`

16.15.17 MultiMarginLoss

class torch.nn.**MultiMarginLoss**(*p=1*, *margin=1.0*, *weight=None*, *size_average=None*, *reduce=None*, *reduction='mean'*)

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, $0 \leq y \leq \text{x.size}(1) - 1$):

For each mini-batch sample, the loss in terms of the 1D input x and scalar output y is:

$$\text{loss}(x, y) = \frac{\sum_i \max(0, \text{margin} - x[y] + x[i]))^p}{\text{x.size}(0)}$$

where $x \in \{0, \cdots, \text{x.size}(0) - 1\}$ and $i \neq y$.

Optionally, you can give non-equal weighting on the classes by passing a 1D `weight` tensor into the constructor.

The loss function then becomes:

$$\text{loss}(x, y) = \frac{\sum_i \max(0, w[y] * (\text{margin} - x[y] + x[i]))^p)}{\text{x.size}(0)}$$

Parameters

- **p** (*int, optional*) – Has a default value of 1. 1 and 2 are the only supported values.
- **margin** (*float, optional*) – Has a default value of 1.
- **weight** (Tensor, *optional*) – a manual rescaling weight given to each class. If given, it has to be a Tensor of size *C*. Otherwise, it is treated as if having all ones.
- **size_average** (*bool, optional*) – Deprecated (see `reduction`). By default, the losses are averaged over each loss element in the batch. Note that for some losses, there are multiple elements per sample. If the field `size_average` is set to `False`, the losses are instead summed for each minibatch. Ignored when reduce is `False`. Default: `True`
- **reduce** (*bool, optional*) – Deprecated (see `reduction`). By default, the losses are averaged or summed over observations for each minibatch depending on `size_average`. When `reduce` is `False`, returns a loss per batch element instead and ignores `size_average`. Default: `True`
- **reduction** (*string, optional*) – Specifies the reduction to apply to the output: `'none'` | `'mean'` | `'sum'`. `'none'`: no reduction will be applied, `'mean'`: the sum of the output will be divided by the number of elements in the output, `'sum'`: the output will be summed. Note: `size_average` and `reduce` are in the process of being deprecated, and in the meantime, specifying either of those two args will override `reduction`. Default: `'mean'`

16.15.18 TripletMarginLoss

class torch.nn.**TripletMarginLoss**(*margin=1.0*, *p=2.0*, *eps=1e-06*, *swap=False*, *size_average=None*, *reduce=None*, *reduction='mean'*)

Creates a criterion that measures the triplet loss given an input tensors $x1$, $x2$, $x3$ and a margin with a value greater than 0. This is used for measuring a relative similarity between samples. A triplet is composed by *a*, *p* and *n* (i.e., *anchor*, *positive examples* and *negative examples* respectively). The shapes of all input tensors should be (N, D).

The distance swap is described in detail in the paper Learning shallow convolutional feature descriptors with triplet losses by V. Balntas, E. Riba et al.

The loss function for each sample in the mini-batch is:

$$L(a, p, n) = \max\{d(a_i, p_i) - d(a_i, n_i) + \text{margin}, 0\}$$

where

$$d(x_i, y_i) = \|\mathbf{x}_i - \mathbf{y}_i\|_p$$

Parameters

- **margin** (*float, optional*) – Default: 1.
- **p** (*int, optional*) – The norm degree for pairwise distance. Default: 2.
- **swap** (*bool, optional*) – The distance swap is described in detail in the paper *Learning shallow convolutional feature descriptors with triplet losses* by V. Balntas, E. Riba et al. Default: `False`.
- **size_average** (*bool, optional*) – Deprecated (see `reduction`). By default, the losses are averaged over each loss element in the batch. Note that for some losses, there are multiple elements per sample. If the field `size_average` is set to `False`, the losses are instead summed for each minibatch. Ignored when reduce is `False`. Default: `True`
- **reduce** (*bool, optional*) – Deprecated (see `reduction`). By default, the losses are averaged or summed over observations for each minibatch depending on `size_average`. When `reduce` is `False`, returns a loss per batch element instead and ignores `size_average`. Default: `True`
- **reduction** (*string, optional*) – Specifies the reduction to apply to the output: `'none'` | `'mean'` | `'sum'`. `'none'`: no reduction will be applied, `'mean'`: the sum of the output will be divided by the number of elements in the output, `'sum'`: the output will be summed. Note: `size_average` and `reduce` are in the process of being deprecated, and in the meantime, specifying either of those two args will override `reduction`. Default: `'mean'`

Shape:

- Input: (N, D) where D is the vector dimension.
- Output: scalar. If `reduction` is `'none'`, then (N).

```
>>> triplet_loss = nn.TripletMarginLoss(margin=1.0, p=2)
>>> input1 = torch.randn(100, 128, requires_grad=True)
>>> input2 = torch.randn(100, 128, requires_grad=True)
>>> input3 = torch.randn(100, 128, requires_grad=True)
>>> output = triplet_loss(input1, input2, input3)
>>> output.backward()
```

16.16 Vision layers

16.16.1 PixelShuffle

class torch.nn.**PixelShuffle**(*upscale_factor*)

Rearranges elements in a tensor of shape $(*, C \times r^2, H, W)$ to a tensor of shape $(*, C, H \times r, W \times r)$.

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.

Parameters **upscale_factor** (*int*) – factor to increase spatial resolution by

Shape:

- Input: (N, L, H_{in}, W_{in}) where $L = C \times \text{upscale\_factor}^2$
- Output: (N, C, H_{out}, W_{out}) where $H_{out} = H_{in} \times \text{upscale\_factor}$ and $W_{out} = W_{in} \times \text{upscale\_factor}$

Examples:

```
>>> pixel_shuffle = nn.PixelShuffle(3)
>>> input = torch.randn(1, 9, 4, 4)
>>> output = pixel_shuffle(input)
>>> print(output.size())
torch.Size([1, 1, 12, 12])
```

16.16.2 Upsample

class torch.nn.**Upsample**(*size=None*, *scale_factor=None*, *mode='nearest'*, *align_corners=None*)

Upsamples a given multi-channel 1D (temporal), 2D (spatial) or 3D (volumetric) data.

The input data is assumed to be of the form *minibatch x channels x [optional depth] x [optional height] x width*. Hence, for spatial inputs, we expect a 4D Tensor and for volumetric inputs, we expect a 5D Tensor.

The algorithms available for upsampling are nearest neighbor and linear, bilinear, bicubic and trilinear for 3D, 4D and 5D input Tensor, respectively.

One can either give a `scale_factor` or the target output `size` to calculate the output size. (You cannot give both, as it is ambiguous)

Parameters

- **size** (*int or Tuple[int] or Tuple[int, int] or Tuple[int, int, int], optional*) – output spatial sizes
- **scale_factor** (*float or Tuple[float] or Tuple[float, float] or Tuple[float, float, float], optional*) – multiplier for spatial size. Has to match input size if it is a tuple.
- **mode** (*str, optional*) – the upsampling algorithm: one of `'nearest'`, `'linear'`, `'bilinear'`, `'bicubic'` and `'trilinear'`. Default: `'nearest'`
- **align_corners** (*bool, optional*) – if `True`, the corner pixels of the input and output tensors are aligned, and thus preserving the values at those pixels. This only has effect when `mode` is `'linear'`, `'bilinear'`, or `'trilinear'`. Default: `False`

Shape:

- Input: (N, C, W_{in}), (N, C, H_{in}, W_{in}) or $(N, C, D_{in}, H_{in}, W_{in})$
- Output: (N, C, W_{out}), (N, C, H_{out}, W_{out}) or $(N, C, D_{out}, H_{out}, W_{out})$, where

$$D_{out} = \lfloor D_{in} \times \text{scale\_factor} \rfloor$$

$$H_{out} = \lfloor H_{in} \times \text{scale\_factor} \rfloor$$

$$W_{out} = \lfloor W_{in} \times \text{scale\_factor} \rfloor$$

Warning: With `align_corners = True`, the linearly interpolating modes (*linear*, *bilinear*, *bicubic*, and *trilinear*) dont proportionally align the output and input pixels, and thus the output values can depend on the input size. This was the default behavior for these modes up to version 0.3.1. Since then, the default behavior is `align_corners = False`. See below for concrete examples on how this affects the outputs.

Note: If you want downsampling/general resizing, you should use `interpolate()`.

Examples:

```
>>> input = torch.arange(1, 5, dtype=torch.float32).view(1, 1, 2, 2)
>>> input
tensor([[[[ 1.,  2.],
          [ 3.,  4.]]]])

>>> m = nn.Upsample(scale_factor=2, mode='nearest')
>>> m(input)
tensor([[[[ 1.,  1.,  2.,  2.],
          [ 1.,  1.,  2.,  2.],
          [ 3.,  3.,  4.,  4.],
          [ 3.,  3.,  4.,  4.]]]])

>>> m = nn.Upsample(scale_factor=2, mode='bilinear')  # align_corners=False
>>> m(input)
tensor([[[[ 1.0000,  1.2500,  1.7500,  2.0000],
          [ 1.5000,  1.7500,  2.2500,  2.5000],
          [ 2.5000,  2.7500,  3.2500,  3.5000],
          [ 3.0000,  3.2500,  3.7500,  4.0000]]]])

>>> m = nn.Upsample(scale_factor=2, mode='bilinear', align_corners=True)
>>> m(input)
tensor([[[[ 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]]]])

>>> # Try scaling the same data in a larger tensor
>>>
>>> input_3x3 = torch.zeros(3, 3).view(1, 1, 3, 3)
>>> input_3x3[:, :, :2, :2].copy_(input)
tensor([[[[ 1.,  2.],
          [ 3.,  4.]]]])
>>> input_3x3
tensor([[[[ 1.,  2.,  0.],
          [ 3.,  4.,  0.],
          [ 0.,  0.,  0.]]]])

>>> m = nn.Upsample(scale_factor=2, mode='bilinear')  # align_corners=False
>>> # Notice that values in top left corner are the same with the small input␣
↪(except at boundary)
>>> m(input_3x3)
tensor([[[[ 1.0000,  1.2500,  1.7500,  1.5000,  0.5000,  0.0000],
          [ 1.5000,  1.7500,  2.2500,  1.8750,  0.6250,  0.0000],
          [ 2.5000,  2.7500,  3.2500,  2.6250,  0.8750,  0.0000],
          [ 2.2500,  2.4375,  2.8125,  2.2500,  0.7500,  0.0000],
```

(continues on next page)

(continued from previous page)

```
          [ 0.7500,  0.8125,  0.9375,  0.7500,  0.2500,  0.0000],
          [ 0.0000,  0.0000,  0.0000,  0.0000,  0.0000,  0.0000]]]])

>>> m = nn.Upsample(scale_factor=2, mode='bilinear', align_corners=True)
>>> # Notice that values in top left corner are now changed
>>> m(input_3x3)
tensor([[[[ 1.0000,  1.4000,  1.8000,  1.6000,  0.8000,  0.0000],
          [ 1.8000,  2.2000,  2.6000,  2.2400,  1.1200,  0.0000],
          [ 2.6000,  3.0000,  3.4000,  2.8800,  1.4400,  0.0000],
          [ 2.4000,  2.7200,  3.0400,  2.5600,  1.2800,  0.0000],
          [ 1.2000,  1.3600,  1.5200,  1.2800,  0.6400,  0.0000],
          [ 0.0000,  0.0000,  0.0000,  0.0000,  0.0000,  0.0000]]]])
```

16.16.3 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 its constructor argument.

When `size` is given, it is the output size of the image *(h, w)*.

Parameters

- **size** (*int or Tuple[int, int], optional*) – output spatial sizes
- **scale_factor** (*float or Tuple[float, float], optional*) – multiplier for spatial size.

Warning: This class is deprecated in favor of `interpolate()`.

Shape:

- Input: (N, C, H_{in}, W_{in})
- Output: (N, C, H_{out}, W_{out}) where

$$H_{out} = \lfloor H_{in} \times \text{scale\_factor} \rfloor$$

$$W_{out} = \lfloor W_{in} \times \text{scale\_factor} \rfloor$$

Examples:

```
>>> input = torch.arange(1, 5, dtype=torch.float32).view(1, 1, 2, 2)
>>> input
tensor([[[[ 1.,  2.],
          [ 3.,  4.]]]])

>>> m = nn.UpsamplingNearest2d(scale_factor=2)
>>> m(input)
tensor([[[[ 1.,  1.,  2.,  2.],
          [ 1.,  1.,  2.,  2.],
          [ 3.,  3.,  4.,  4.],
          [ 3.,  3.,  4.,  4.]]]])
```

16.16.4 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 its constructor argument.

When `size` is given, it is the output size of the image *(h, w)*.

Parameters

- **size** (*int or Tuple[int, int], optional*) – output spatial sizes
- **scale_factor** (*float or Tuple[float, float], optional*) – multiplier for spatial size.

Warning: This class is deprecated in favor of `interpolate()`. It is equivalent to `nn.functional.interpolate(..., mode='bilinear', align_corners=True)`.

Shape:

- Input: (N, C, H_{in}, W_{in})
- Output: (N, C, H_{out}, W_{out}) where

$$H_{out} = \lfloor H_{in} \times \text{scale\_factor} \rfloor$$

$$W_{out} = \lfloor W_{in} \times \text{scale\_factor} \rfloor$$

Examples:

```
>>> input = torch.arange(1, 5, dtype=torch.float32).view(1, 1, 2, 2)
>>> input
tensor([[[[ 1.,  2.],
          [ 3.,  4.]]]])

>>> m = nn.UpsamplingBilinear2d(scale_factor=2)
>>> m(input)
tensor([[[[ 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]]]])
```

16.17 DataParallel layers (multi-GPU, distributed)

16.17.1 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 (other objects will be copied once per device). 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.

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

Arbitrary positional and keyword inputs are allowed to be passed into DataParallel but some types are specially handled. tensors will be **scattered** on dim specified (default 0). tuple, list and dict types will be shallow copied. The other types will be shared among different threads and can be corrupted if written to in the models forward pass.

The parallelized `module` must have its parameters and buffers on `device_ids[0]` before running this *`DataParallel`* module.

Warning: In each forward, `module` is **replicated** on each device, so any updates to the running module in `forward` will be lost. For example, if `module` has a counter attribute that is incremented in each `forward`, it will always stay at the initial value because the update is done on the replicas which are destroyed after `forward`. However, *`DataParallel`* guarantees that the replica on `device[0]` will have its parameters and buffers sharing storage with the base parallelized `module`. So **in-place** updates to the parameters or buffers on `device[0]` will be recorded. E.g., *`BatchNorm2d`* and *`spectral_norm()`* rely on this behavior to update the buffers.

Warning: Forward and backward hooks defined on `module` and its submodules will be invoked `len(device_ids)` times, each with inputs located on a particular device. Particularly, the hooks are only guaranteed to be executed in correct order with respect to operations on corresponding devices. For example, it is not guaranteed that hooks set via *`register_forward_pre_hook()`* be executed before *all* `len(device_ids)` *`forward()`* calls, but that each such hook be executed before the corresponding *`forward()`* call of that device.

Warning: When `module` returns a scalar (i.e., 0-dimensional tensor) in `forward()`, this wrapper will return a vector of length equal to number of devices used in data parallelism, containing the result from each device.

Note: There is a subtlety in using the `pack sequence -> recurrent network -> unpack sequence` pattern in a *`Module`* wrapped in *`DataParallel`*. See *My recurrent network doesnt work with data parallelism* section in FAQ for details.

Parameters

- **`module`** (`Module`) – module to be parallelized
- **`device_ids`** (*`list of python:int or torch.device`*) – CUDA devices (default: all devices)
- **`output_device`** (*`int or torch.device`*) – device location of output (default: device_ids[0])

Variables **`module`** (`Module`) – the module to be parallelized

Example:

```
>>> net = torch.nn.DataParallel(model, device_ids=[0, 1, 2])
>>> output = net(input_var)  # input_var can be on any device, including CPU
```

16.17.2 DistributedDataParallel

class torch.nn.parallel.**DistributedDataParallel**(*module*, *device_ids=None*, *output_device=None*, *dim=0*, *broadcast_buffers=True*, *process_group=None*, *bucket_cap_mb=25*, *check_reduction=False*)

Implements distributed data parallelism that is based on `torch.distributed` package 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. The module is replicated on each machine and each device, and each such replica handles a portion of the input. During the backwards pass, gradients from each node are averaged.

The batch size should be larger than the number of GPUs used locally.

See also: *Basics* and *Use nn.DataParallel instead of multiprocessing*. The same constraints on input as in *torch.nn.DataParallel* apply.

Creation of this class requires that `torch.distributed` to be already initialized, by calling *torch.distributed.init_process_group()*.

`DistributedDataParallel` can be used in the following two ways:

1. Single-Process Multi-GPU

In this case, a single process will be spawned on each host/node and each process will operate on all the GPUs of the node where its running. To use `DistributedDataParallel` in this way, you can simply construct the model as the following:

```
>>> torch.distributed.init_process_group(backend="nccl")
>>> model = DistributedDataParallel(model) # device_ids will include all GPU␣
↪devices by default
```

2. Multi-Process Single-GPU

This is the highly recommended way to use `DistributedDataParallel`, with multiple processes, each of which operates on a single GPU. This is currently the fastest approach to do data parallel training using PyTorch and applies to both single-node(multi-GPU) and multi-node data parallel training. It is proven to be significantly faster than *torch.nn.DataParallel* for single-node multi-GPU data parallel training.

Here is how to use it: on each host with N GPUs, you should spawn up N processes, while ensuring that each process individually works on a single GPU from 0 to N-1. Therefore, it is your job to ensure that your training script operates on a single given GPU by calling:

```
>>> torch.cuda.set_device(i)
```

where i is from 0 to N-1. In each process, you should refer the following to construct this module:

```
>>> torch.distributed.init_process_group(backend='nccl', world_size=4, init_
↪method='...')
>>> model = DistributedDataParallel(model, device_ids=[i], output_device=i)
```

In order to spawn up multiple processes per node, you can use either `torch.distributed.launch` or `torch.multiprocessing.spawn`

Note: `nccl` backend is currently the fastest and highly recommended backend to be used with Multi-Process

Single-GPU distributed training and this applies to both single-node and multi-node distributed training

Note: This module also supports mixed-precision distributed training. This means that your model can have different types of parameters such as mixed types of fp16 and fp32, the gradient reduction on these mixed types of parameters will just work fine. Also note that `nccl` backend is currently the fastest and highly recommended backend for fp16/fp32 mixed-precision training.

Warning: This module works only with the `gloo` and `nccl` backends.

Warning: Constructor, forward method, and differentiation of the output (or a function of the output of this module) is a distributed synchronization point. Take that into account in case different processes might be executing different code.

Warning: This module assumes all parameters are registered in the model by the time it is created. No parameters should be added nor removed later. Same applies to buffers.

Warning: This module assumes all parameters are registered in the model of each distributed processes are in the same order. The module itself will conduct gradient all-reduction following the reverse order of the registered parameters of the model. In other words, it is users responsibility to ensure that each distributed process has the exact same model and thus the exact same parameter registration order.

Warning: This module assumes all buffers and gradients are dense.

Warning: This module doesnt work with *`torch.autograd.grad()`* (i.e. it will only work if gradients are to be accumulated in `.grad` attributes of parameters).

Warning: If you plan on using this module with a `nccl` backend or a `gloo` backend (that uses Infiniband), together with a DataLoader that uses multiple workers, please change the multiprocessing start method to `forkserver` (Python 3 only) or `spawn`. Unfortunately Gloo (that uses Infiniband) and NCCL2 are not fork safe, and you will likely experience deadlocks if you dont change this setting.

Warning: Forward and backward hooks defined on `module` and its submodules wont be invoked anymore, unless the hooks are initialized in the `forward()` method.

Warning: You should never try to change your models parameters after wrapping up your model with DistributedDataParallel. In other words, when wrapping up your model with DistributedDataParallel, the constructor of DistributedDataParallel will register the additional gradient reduction functions on all the parameters of the model itself at the time of construction. If you change the models parameters after the DistributedDataParallel construction, this is not supported and unexpected behaviors can happen, since some parameters gradient reduction functions might not get called.

Note: Parameters are never broadcast between processes. The module performs an all-reduce step on gradients and assumes that they will be modified by the optimizer in all processes in the same way. Buffers (e.g. BatchNorm stats) are broadcast from the module in process of rank 0, to all other replicas in the system in every iteration.

Parameters

- **module** (Module) – module to be parallelized
- **device_ids** (*list of python:int or* torch.device) – CUDA devices (default: all devices)
- **output_device** (*int or* torch.device) – device location of output (default: device_ids[0])
- **broadcast_buffers** (*bool*) – flag that enables syncing (broadcasting) buffers of the module at beginning of the forward function. (default: True)
- **process_group** – the process group to be used for distributed data all-reduction. If None, the default process group, which is created by `torch.distributed.init_process_group`, will be used. (default: None)
- **bucket_cap_mb** – DistributedDataParallel will bucket parameters into multiple buckets so that gradient reduction of each bucket can potentially overlap with backward computation. bucket_cap_mb controls the bucket size in MegaBytes (MB) (default: 25)
- **check_reduction** – when setting to True, it enables DistributedDataParallel to automatically check if the previous iterations backward reductions were successfully issued at the beginning of every iterations forward function. You normally dont need this option enabled unless you are observing weird behaviors such as different ranks are getting different gradients, which should not happen if DistributedDataParallel is correctly used. (default: False)

Variables **module** (Module) – the module to be parallelized

Example:

```
>>> torch.distributed.init_process_group(backend='nccl', world_size=4, init_
↪method='...')
>>> net = torch.nn.DistributedDataParallel(model, pg)
```

16.18 Utilities

16.18.1 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

- **parameters** (*Iterable[Tensor] or Tensor*) – an iterable of Tensors or a single Tensor 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.

Returns Total norm of the parameters (viewed as a single vector).

16.18.2 clip_grad_value_

torch.nn.utils.**clip_grad_value_**(*parameters*, *clip_value*)

Clips gradient of an iterable of parameters at specified value.

Gradients are modified in-place.

Parameters

- **parameters** (*Iterable[Tensor] or Tensor*) – an iterable of Tensors or a single Tensor that will have gradients normalized
- **clip_value** (*float or int*) – maximum allowed value of the gradients. The gradients are clipped in the range [-clip_value, clip_value]

16.18.3 parameters_to_vector

torch.nn.utils.**parameters_to_vector**(*parameters*)

Convert parameters to one vector

Parameters **parameters** (*Iterable[Tensor]*) – an iterator of Tensors that are the parameters of a model.

Returns The parameters represented by a single vector

16.18.4 vector_to_parameters

torch.nn.utils.**vector_to_parameters**(*vec*, *parameters*)

Convert one vector to the parameters

Parameters

- **vec** (Tensor) – a single vector represents the parameters of a model.
- **parameters** (*Iterable[Tensor]*) – an iterator of Tensors that are the parameters of a model.

16.18.5 weight_norm

torch.nn.utils.**weight_norm**(*module*, *name='weight'*, *dim=0*)

Applies weight normalization to a parameter in the given module.

$$\mathbf{w} = g\frac{\mathbf{v}}{\|\mathbf{v}\|}$$

Weight normalization is a reparameterization that decouples the magnitude of a weight tensor from its direction. This replaces the parameter specified by `name` (e.g. `'weight'`) with two parameters: one specifying the magnitude (e.g. `'weight_g'`) and one specifying the direction (e.g. `'weight_v'`). Weight normalization is implemented via a hook that recomputes the weight tensor from the magnitude and direction before every `forward()` call.

By default, with `dim=0`, the norm is computed independently per output channel/plane. To compute a norm over the entire weight tensor, use `dim=None`.

See https://arxiv.org/abs/1602.07868

Parameters

- **module** (Module) – containing module
- **name** (*str, optional*) – name of weight parameter
- **dim** (*int, optional*) – dimension over which to compute the norm

Returns The original module with the weight norm hook

Example:

```
>>> m = weight_norm(nn.Linear(20, 40), name='weight')
>>> m
Linear(in_features=20, out_features=40, bias=True)
>>> m.weight_g.size()
torch.Size([40, 1])
>>> m.weight_v.size()
torch.Size([40, 20])
```

16.18.6 remove_weight_norm

torch.nn.utils.**remove_weight_norm**(*module*, *name='weight'*)

Removes the weight normalization reparameterization from a module.

Parameters

- **module** (Module) – containing module
- **name** (*str, optional*) – name of weight parameter

Example

```
>>> m = weight_norm(nn.Linear(20, 40))
>>> remove_weight_norm(m)
```

16.18.7 spectral_norm

torch.nn.utils.**spectral_norm**(*module*, *name='weight'*, *n_power_iterations=1*, *eps=1e-12*, *dim=None*)

Applies spectral normalization to a parameter in the given module.

$$\mathbf{W}_{SN} = \frac{\mathbf{W}}{\sigma(\mathbf{W})}, \sigma(\mathbf{W}) = \max_{\mathbf{h}:\mathbf{h}\neq 0} \frac{\|\mathbf{W}\mathbf{h}\|_2}{\|\mathbf{h}\|_2}$$

Spectral normalization stabilizes the training of discriminators (critics) in Generative Adversarial Networks (GANs) by rescaling the weight tensor with spectral norm σ of the weight matrix calculated using power iteration method. If the dimension of the weight tensor is greater than 2, it is reshaped to 2D in power iteration method to get spectral norm. This is implemented via a hook that calculates spectral norm and rescales weight before every `forward()` call.

See Spectral Normalization for Generative Adversarial Networks .

Parameters

- **module** (nn.Module) – containing module
- **name** (*str, optional*) – name of weight parameter
- **n_power_iterations** (*int, optional*) – number of power iterations to calculate spectral norm
- **eps** (*float, optional*) – epsilon for numerical stability in calculating norms
- **dim** (*int, optional*) – dimension corresponding to number of outputs, the default is 0, except for modules that are instances of ConvTranspose{1,2,3}d, when it is 1

Returns The original module with the spectral norm hook

Example:

```
>>> m = spectral_norm(nn.Linear(20, 40))
>>> m
Linear(in_features=20, out_features=40, bias=True)
>>> m.weight_u.size()
torch.Size([40])
```

16.18.8 remove_spectral_norm

torch.nn.utils.**remove_spectral_norm**(*module*, *name='weight'*)

Removes the spectral normalization reparameterization from a module.

Parameters

- **module** (Module) – containing module
- **name** (*str, optional*) – name of weight parameter

Example

```
>>> m = spectral_norm(nn.Linear(40, 10))
>>> remove_spectral_norm(m)
```

16.18.9 PackedSequence

torch.nn.utils.rnn.**PackedSequence**(*data*, *batch_sizes=None*, *sorted_indices=None*, *unsorted_indices=None*)

Holds the data and list of `batch_sizes` of a packed sequence.

All RNN modules accept packed sequences as inputs.

Note: Instances of this class should never be created manually. They are meant to be instantiated by functions like *pack_padded_sequence()*.

Batch sizes represent the number elements at each sequence step in the batch, not the varying sequence lengths passed to *pack_padded_sequence()*. For instance, given data `abc` and `x` the *PackedSequence* would contain data `axbc` with `batch_sizes=[2,1,1]`.

Variables

- **data** (Tensor) – Tensor containing packed sequence
- **batch_sizes** (Tensor) – Tensor of integers holding information about the batch size at each sequence step
- **sorted_indices** (Tensor, *optional*) – Tensor of integers holding how this *PackedSequence* is constructed from sequences.
- **unsorted_indices** (Tensor, *optional*) – Tensor of integers holding how this to recover the original sequences with correct order.

Note: `data` can be on arbitrary device and of arbitrary dtype. `sorted_indices` and `unsorted_indices` must be `torch.int64` tensors on the same device as `data`.

However, `batch_sizes` should always be a CPU `torch.int64` tensor.

This invariant is maintained throughout *PackedSequence* class, and all functions that construct a *:class:PackedSequence* in PyTorch (i.e., they only pass in tensors conforming to this constraint).

16.18.10 pack_padded_sequence

torch.nn.utils.rnn.**pack_padded_sequence**(*input*, *lengths*, *batch_first=False*, *enforce_sorted=True*)

Packs a Tensor containing padded sequences of variable length.

`input` can be of size `T x B x *` 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`, `B x T x *` `input` is expected.

For unsorted sequences, use *enforce_sorted* = *False*. If `enforce_sorted` is `True`, 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. *enforce_sorted* = *True* is only necessary for ONNX export.

Note: This function accepts 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 Tensor can be retrieved from a *PackedSequence* object by accessing its `.data` attribute.

Parameters

- **input** (Tensor) – padded batch of variable length sequences.
- **lengths** (Tensor) – list of sequences lengths of each batch element.
- **batch_first** (*bool, optional*) – if `True`, the input is expected in `B x T x *` format.
- **enforce_sorted** (*bool, optional*) – if `True`, the input is expected to contain sequences sorted by length in a decreasing order. If `False`, this condition is not checked. Default: `True`.

Returns a *PackedSequence* object

16.18.11 pad_packed_sequence

torch.nn.utils.rnn.**pad_packed_sequence**(*sequence*, *batch_first=False*, *padding_value=0.0*, *total_length=None*)

Pads a packed batch of variable length sequences.

It is an inverse operation to *pack_padded_sequence()*.

The returned Tensors data will be of size `T x B x *`, where *T* is the length of the longest sequence and *B* is the batch size. If `batch_first` is True, the data will be transposed into `B x T x *` format.

Batch elements will be ordered decreasingly by their length.

Note: `total_length` is useful to implement the `pack sequence -> recurrent network -> unpack sequence` pattern in a *Module* wrapped in *DataParallel*. See *this FAQ section* for details.

Parameters

- **sequence** (*PackedSequence*) – batch to pad
- **batch_first** (*bool, optional*) – if `True`, the output will be in `B x T x *` format.
- **padding_value** (*float, optional*) – values for padded elements.
- **total_length** (*int, optional*) – if not `None`, the output will be padded to have length `total_length`. This method will throw `ValueError` if `total_length` is less than the max sequence length in `sequence`.

Returns Tuple of Tensor containing the padded sequence, and a Tensor containing the list of lengths of each sequence in the batch.

16.18.12 pad_sequence

torch.nn.utils.rnn.**pad_sequence**(*sequences*, *batch_first=False*, *padding_value=0*)

Pad a list of variable length Tensors with `padding_value`

`pad_sequence` stacks a list of Tensors along a new dimension, and pads them to equal length. For example, if the input is list of sequences with size `L x *` and if batch_first is False, and `T x B x *` otherwise.

B is batch size. It is equal to the number of elements in `sequences`. *T* is length of the longest sequence. *L* is length of the sequence. * is any number of trailing dimensions, including none.

Example

```
>>> from torch.nn.utils.rnn import pad_sequence
>>> a = torch.ones(25, 300)
>>> b = torch.ones(22, 300)
>>> c = torch.ones(15, 300)
>>> pad_sequence([a, b, c]).size()
torch.Size([25, 3, 300])
```

Note: This function returns a Tensor of size `T x B x *` or `B x T x *` where *T* is the length of the longest sequence. This function assumes trailing dimensions and type of all the Tensors in sequences are same.

Parameters

- **sequences** (*list[Tensor]*) – list of variable length sequences.
- **batch_first** (*bool, optional*) – output will be in `B x T x *` if True, or in `T x B x *` otherwise
- **padding_value** (*float, optional*) – value for padded elements. Default: 0.

Returns Tensor of size `T x B x *` if `batch_first` is `False`. Tensor of size `B x T x *` otherwise

16.18.13 pack_sequence

`torch.nn.utils.rnn.`**pack_sequence**(*sequences*, *enforce_sorted=True*)

Packs a list of variable length Tensors

`sequences` should be a list of Tensors of size `L x *`, where *L* is the length of a sequence and * is any number of trailing dimensions, including zero.

For unsorted sequences, use *enforce_sorted* = *False*. If `enforce_sorted` is `True`, the sequences should be sorted in the order of decreasing length. `enforce_sorted = True` is only necessary for ONNX export.

Example

```
>>> from torch.nn.utils.rnn import pack_sequence
>>> a = torch.tensor([1,2,3])
>>> b = torch.tensor([4,5])
>>> c = torch.tensor([6])
>>> pack_sequence([a, b, c])
PackedSequence(data=tensor([ 1,  4,  6,  2,  5,  3]), batch_sizes=tensor([ 3,  2,␣
↪ 1]))
```

Parameters

- **sequences** (*list[Tensor]*) – A list of sequences of decreasing length.
- **enforce_sorted** (*bool, optional*) – if `True`, checks that the input contains sequences sorted by length in a decreasing order. If `False`, this condition is not checked. Default: `True`.

Returns a *PackedSequence* object

16.18.14 Flatten

16.19 Quantized Functions

Quantization refers to techniques for performing computations and storing tensors at lower bitwidths than floating point precision. PyTorch supports both per tensor and per channel asymmetric linear quantization. To learn more how to use quantized functions in PyTorch, please refer to the *Quantization* documentation.

CHAPTER

SEVENTEEN

TORCH.NN.FUNCTIONAL

17.1 Convolution functions

17.1.1 conv1d

torch.nn.functional.**conv1d**(*input*, *weight*, *bias=None*, *stride=1*, *padding=0*, *dilation=1*, *groups=1*, *padding_mode='zeros'*) → Tensor

Applies a 1D convolution over an input signal composed of several input planes.

See *Conv1d* for details and output shape.

Note: In some circumstances when using the CUDA backend with CuDNN, this operator may select a nondeterministic algorithm to increase performance. If this is undesirable, you can try to make the operation deterministic (potentially at a performance cost) by setting `torch.backends.cudnn.deterministic = True`. Please see the notes on *Reproducibility* for background.

Parameters

- **input** – input tensor of shape $(\text{minibatch}, \text{in\_channels}, iW)$
- **weight** – filters of shape $(\text{out\_channels}, \frac{\text{in\_channels}}{\text{groups}}, kW)$
- **bias** – optional bias of shape (out_channels). Default: `None`
- **stride** – the stride of the convolving kernel. Can be a single number or a one-element tuple *(sW,)*. Default: 1
- **padding** – implicit paddings on both sides of the input. Can be a single number or a one-element tuple *(padW,)*. Default: 0
- **dilation** – the spacing between kernel elements. Can be a single number or a one-element tuple *(dW,)*. Default: 1
- **groups** – split input into groups, in_channels should be divisible by the number of groups. Default: 1
- **padding_mode** – the type of paddings applied to both sided can be: *zeros* or *circular*. Default: *zeros*

Examples:

```
>>> filters = torch.randn(33, 16, 3)
>>> inputs = torch.randn(20, 16, 50)
>>> F.conv1d(inputs, filters)
```

17.1.2 conv2d

torch.nn.functional.**conv2d**(*input*, *weight*, *bias=None*, *stride=1*, *padding=0*, *dilation=1*, *groups=1*, *padding_mode='zeros'*) → Tensor

Applies a 2D convolution over an input image composed of several input planes.

See *Conv2d* for details and output shape.

Note: In some circumstances when using the CUDA backend with CuDNN, this operator may select a nondeterministic algorithm to increase performance. If this is undesirable, you can try to make the operation deterministic (potentially at a performance cost) by setting `torch.backends.cudnn.deterministic = True`. Please see the notes on *Reproducibility* for background.

Parameters

- **input** – input tensor of shape $(\text{minibatch}, \text{in\_channels}, iH, iW)$
- **weight** – filters of shape $(\text{out\_channels}, \frac{\text{in\_channels}}{\text{groups}}, kH, kW)$
- **bias** – optional bias tensor of shape (out_channels). Default: `None`
- **stride** – the stride of the convolving kernel. Can be a single number or a tuple *(sH, sW)*. Default: 1
- **padding** – implicit paddings on both sides of the input. Can be a single number or a tuple *(padH, padW)*. Default: 0
- **dilation** – the spacing between kernel elements. Can be a single number or a tuple *(dH, dW)*. Default: 1
- **groups** – split input into groups, in_channels should be divisible by the number of groups. Default: 1
- **padding_mode** – the type of paddings applied to both sided can be: *zeros* or *circular*. Default: *zeros*

Examples:

```
>>> # With square kernels and equal stride
>>> filters = torch.randn(8,4,3,3)
>>> inputs = torch.randn(1,4,5,5)
>>> F.conv2d(inputs, filters, padding=1)
```

17.1.3 conv3d

torch.nn.functional.**conv3d**(*input*, *weight*, *bias=None*, *stride=1*, *padding=0*, *dilation=1*, *groups=1*, *padding_mode='zeros'*) → Tensor

Applies a 3D convolution over an input image composed of several input planes.

See *Conv3d* for details and output shape.

Note: In some circumstances when using the CUDA backend with CuDNN, this operator may select a nondeterministic algorithm to increase performance. If this is undesirable, you can try to make the operation deterministic (potentially at a performance cost) by setting `torch.backends.cudnn.deterministic = True`. Please see the notes on *Reproducibility* for background.

Parameters

- **input** – input tensor of shape $(\text{minibatch}, \text{in\_channels}, iT, iH, iW)$
- **weight** – filters of shape $(\text{out\_channels}, \frac{\text{in\_channels}}{\text{groups}}, kT, kH, kW)$
- **bias** – optional bias tensor of shape (out_channels). Default: None
- **stride** – the stride of the convolving kernel. Can be a single number or a tuple *(sT, sH, sW)*. Default: 1
- **padding** – implicit paddings on both sides of the input. Can be a single number or a tuple *(padT, padH, padW)*. Default: 0
- **dilation** – the spacing between kernel elements. Can be a single number or a tuple *(dT, dH, dW)*. Default: 1
- **groups** – split input into groups, in_channels should be divisible by the number of groups. Default: 1
- **padding_mode** – the type of paddings applied to both sided can be: *zeros* or *circular*. Default: *zeros*

Examples:

```
>>> filters = torch.randn(33, 16, 3, 3, 3)
>>> inputs = torch.randn(20, 16, 50, 10, 20)
>>> F.conv3d(inputs, filters)
```

17.1.4 conv_transpose1d

torch.nn.functional.**conv_transpose1d**(*input*, *weight*, *bias=None*, *stride=1*, *padding=0*, *output_padding=0*, *groups=1*, *dilation=1*) → Tensor

Applies a 1D transposed convolution operator over an input signal composed of several input planes, sometimes also called deconvolution.

See *ConvTranspose1d* for details and output shape.

Note: In some circumstances when using the CUDA backend with CuDNN, this operator may select a nondeterministic algorithm to increase performance. If this is undesirable, you can try to make the operation deterministic (potentially at a performance cost) by setting `torch.backends.cudnn.deterministic = True`. Please see the notes on *Reproducibility* for background.

Parameters

- **input** – input tensor of shape $(\text{minibatch}, \text{in\_channels}, iW)$
- **weight** – filters of shape $(\text{in\_channels}, \frac{\text{out\_channels}}{\text{groups}}, kW)$
- **bias** – optional bias of shape (out_channels). Default: None
- **stride** – the stride of the convolving kernel. Can be a single number or a tuple `(sW,)`. Default: 1
- **padding** – `dilation * (kernel_size - 1) - padding` zero-padding will be added to both sides of each dimension in the input. Can be a single number or a tuple `(padW,)`. Default: 0

- **output_padding** – additional size added to one side of each dimension in the output shape. Can be a single number or a tuple `(out_padW)`. Default: 0
- **groups** – split input into groups, in_channels should be divisible by the number of groups. Default: 1
- **dilation** – the spacing between kernel elements. Can be a single number or a tuple `(dW,)`. Default: 1

Examples:

```
>>> inputs = torch.randn(20, 16, 50)
>>> weights = torch.randn(16, 33, 5)
>>> F.conv_transpose1d(inputs, weights)
```

17.1.5 conv_transpose2d

torch.nn.functional.**conv_transpose2d**(*input*, *weight*, *bias=None*, *stride=1*, *padding=0*, *output_padding=0*, *groups=1*, *dilation=1*) → Tensor

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.

Note: In some circumstances when using the CUDA backend with CuDNN, this operator may select a nondeterministic algorithm to increase performance. If this is undesirable, you can try to make the operation deterministic (potentially at a performance cost) by setting `torch.backends.cudnn.deterministic = True`. Please see the notes on *Reproducibility* for background.

Parameters

- **input** – input tensor of shape $(\text{minibatch}, \text{in\_channels}, iH, iW)$
- **weight** – filters of shape $(\text{in\_channels}, \frac{\text{out\_channels}}{\text{groups}}, kH, kW)$
- **bias** – optional bias of shape (out_channels). Default: None
- **stride** – the stride of the convolving kernel. Can be a single number or a tuple `(sH, sW)`. Default: 1
- **padding** – `dilation * (kernel_size - 1) - padding` zero-padding will be added to both sides of each dimension in the input. Can be a single number or a tuple `(padH, padW)`. Default: 0
- **output_padding** – additional size added to one side of each dimension in the output shape. Can be a single number or a tuple `(out_padH, out_padW)`. Default: 0
- **groups** – split input into groups, in_channels should be divisible by the number of groups. Default: 1
- **dilation** – the spacing between kernel elements. Can be a single number or a tuple `(dH, dW)`. Default: 1

Examples:

```
>>> # With square kernels and equal stride
>>> inputs = torch.randn(1, 4, 5, 5)
>>> weights = torch.randn(4, 8, 3, 3)
>>> F.conv_transpose2d(inputs, weights, padding=1)
```

17.1.6 conv_transpose3d

torch.nn.functional.**conv_transpose3d**(*input*, *weight*, *bias=None*, *stride=1*, *padding=0*, *output_padding=0*, *groups=1*, *dilation=1*) → Tensor

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.

Note: In some circumstances when using the CUDA backend with CuDNN, this operator may select a nondeterministic algorithm to increase performance. If this is undesirable, you can try to make the operation deterministic (potentially at a performance cost) by setting `torch.backends.cudnn.deterministic = True`. Please see the notes on *Reproducibility* for background.

Parameters

- **input** – input tensor of shape $(\text{minibatch}, \text{in\_channels}, iT, iH, iW)$
- **weight** – filters of shape $(\text{in\_channels}, \frac{\text{out\_channels}}{\text{groups}}, kT, kH, kW)$
- **bias** – optional bias of shape (out_channels). Default: None
- **stride** – the stride of the convolving kernel. Can be a single number or a tuple `(sT, sH, sW)`. Default: 1
- **padding** – `dilation * (kernel_size - 1) - padding` zero-padding will be added to both sides of each dimension in the input. Can be a single number or a tuple `(padT, padH, padW)`. Default: 0
- **output_padding** – additional size added to one side of each dimension in the output shape. Can be a single number or a tuple `(out_padT, out_padH, out_padW)`. Default: 0
- **groups** – split input into groups, in_channels should be divisible by the number of groups. Default: 1
- **dilation** – the spacing between kernel elements. Can be a single number or a tuple *(dT, dH, dW)*. Default: 1

Examples:

```
>>> inputs = torch.randn(20, 16, 50, 10, 20)
>>> weights = torch.randn(16, 33, 3, 3, 3)
>>> F.conv_transpose3d(inputs, weights)
```

17.1.7 unfold

torch.nn.functional.**unfold**(*input*, *kernel_size*, *dilation=1*, *padding=0*, *stride=1*)

Extracts sliding local blocks from an batched input tensor.

> **Warning:** Currently, only 4-D input tensors (batched image-like tensors) are supported.

> **Warning:** More than one element of the unfolded tensor may refer to a single memory location. As a result, in-place operations (especially ones that are vectorized) may result in incorrect behavior. If you need to write to the tensor, please clone it first.

See *torch.nn.Unfold* for details

17.1.8 fold

torch.nn.functional.**fold**(*input*, *output_size*, *kernel_size*, *dilation=1*, *padding=0*, *stride=1*)

Combines an array of sliding local blocks into a large containing tensor.

> **Warning:** Currently, only 4-D output tensors (batched image-like tensors) are supported.

See *torch.nn.Fold* for details

17.2 Pooling functions

17.2.1 avg_pool1d

torch.nn.functional.**avg_pool1d**(*input*, *kernel_size*, *stride=None*, *padding=0*, *ceil_mode=False*, *count_include_pad=True*) → Tensor

Applies a 1D average pooling over an input signal composed of several input planes.

See *AvgPool1d* for details and output shape.

Parameters

- **input** – input tensor of shape $(\text{minibatch}, \text{in\_channels}, iW)$
- **kernel_size** – the size of the window. Can be a single number or a tuple *(kW,)*
- **stride** – the stride of the window. Can be a single number or a tuple *(sW,)*. Default: `kernel_size`
- **padding** – implicit zero paddings on both sides of the input. Can be a single number or a tuple *(padW,)*. Default: 0
- **ceil_mode** – when True, will use *ceil* instead of *floor* to compute the output shape. Default: `False`
- **count_include_pad** – when True, will include the zero-padding in the averaging calculation. Default: `True`

Examples:

```
>>> # pool of square window of size=3, stride=2
>>> input = torch.tensor([[[1, 2, 3, 4, 5, 6, 7]]], dtype=torch.float32)
>>> F.avg_pool1d(input, kernel_size=3, stride=2)
tensor([[[ 2.,  4.,  6.]]])
```

17.2.2 avg_pool2d

torch.nn.functional.**avg_pool2d**(*input*, *kernel_size*, *stride=None*, *padding=0*, *ceil_mode=False*, *count_include_pad=True*) → Tensor

Applies 2D average-pooling operation in $kH \times kW$ regions by step size $sH \times sW$ steps. The number of output features is equal to the number of input planes.

See `AvgPool2d` for details and output shape.

Parameters

- **input** – input tensor $(\text{minibatch}, \text{in\_channels}, iH, iW)$
- **kernel_size** – size of the pooling region. Can be a single number or a tuple *(kH, kW)*
- **stride** – stride of the pooling operation. Can be a single number or a tuple *(sH, sW)*. Default: `kernel_size`
- **padding** – implicit zero paddings on both sides of the input. Can be a single number or a tuple *(padH, padW)*. Default: 0
- **ceil_mode** – when True, will use *ceil* instead of *floor* in the formula to compute the output shape. Default: `False`
- **count_include_pad** – when True, will include the zero-padding in the averaging calculation. Default: `True`

17.2.3 avg_pool3d

torch.nn.functional.**avg_pool3d**(*input*, *kernel_size*, *stride=None*, *padding=0*, *ceil_mode=False*, *count_include_pad=True*) → Tensor

Applies 3D average-pooling operation in $kT \times kH \times kW$ regions by step size $sT \times sH \times sW$ steps. The number of output features is equal to $\lfloor \frac{\text{input planes}}{sT} \rfloor$.

See `AvgPool3d` for details and output shape.

Parameters

- **input** – input tensor $(\text{minibatch}, \text{in\_channels}, iT \times iH, iW)$
- **kernel_size** – size of the pooling region. Can be a single number or a tuple *(kT, kH, kW)*
- **stride** – stride of the pooling operation. Can be a single number or a tuple *(sT, sH, sW)*. Default: `kernel_size`
- **padding** – implicit zero paddings on both sides of the input. Can be a single number or a tuple *(padT, padH, padW)*, Default: 0
- **ceil_mode** – when True, will use *ceil* instead of *floor* in the formula to compute the output shape
- **count_include_pad** – when True, will include the zero-padding in the averaging calculation

17.2.4 max_pool1d

torch.nn.functional.**max_pool1d**(**args*, ***kwargs*)

Applies a 1D max pooling over an input signal composed of several input planes.

See `MaxPool1d` for details.

17.2.5 max_pool2d

torch.nn.functional.**max_pool2d**(**args*, ***kwargs*)
Applies a 2D max pooling over an input signal composed of several input planes.

See *MaxPool2d* for details.

17.2.6 max_pool3d

torch.nn.functional.**max_pool3d**(**args*, ***kwargs*)
Applies a 3D max pooling over an input signal composed of several input planes.

See *MaxPool3d* for details.

17.2.7 max_unpool1d

torch.nn.functional.**max_unpool1d**(*input*, *indices*, *kernel_size*, *stride=None*, *padding=0*, *output_size=None*)
Computes a partial inverse of MaxPool1d.

See *MaxUnpool1d* for details.

17.2.8 max_unpool2d

torch.nn.functional.**max_unpool2d**(*input*, *indices*, *kernel_size*, *stride=None*, *padding=0*, *output_size=None*)
Computes a partial inverse of MaxPool2d.

See *MaxUnpool2d* for details.

17.2.9 max_unpool3d

torch.nn.functional.**max_unpool3d**(*input*, *indices*, *kernel_size*, *stride=None*, *padding=0*, *output_size=None*)
Computes a partial inverse of MaxPool3d.

See *MaxUnpool3d* for details.

17.2.10 lp_pool1d

torch.nn.functional.**lp_pool1d**(*input*, *norm_type*, *kernel_size*, *stride=None*, *ceil_mode=False*)
Applies a 1D power-average pooling over an input signal composed of several input planes. If the sum of all inputs to the power of *p* is zero, the gradient is set to zero as well.

See *LPPool1d* for details.

17.2.11 lp_pool2d

torch.nn.functional.**lp_pool2d**(*input*, *norm_type*, *kernel_size*, *stride=None*, *ceil_mode=False*)
Applies a 2D power-average pooling over an input signal composed of several input planes. If the sum of all inputs to the power of *p* is zero, the gradient is set to zero as well.

See *LPPool2d* for details.

17.2.12 adaptive_max_pool1d

torch.nn.functional.**adaptive_max_pool1d**(**args*, ***kwargs*)
Applies a 1D adaptive max pooling over an input signal composed of several input planes.

See *AdaptiveMaxPool1d* for details and output shape.

Parameters

- **output_size** – the target output size (single integer)
- **return_indices** – whether to return pooling indices. Default: False

17.2.13 adaptive_max_pool2d

torch.nn.functional.**adaptive_max_pool2d**(**args*, ***kwargs*)
Applies a 2D adaptive max pooling over an input signal composed of several input planes.

See *AdaptiveMaxPool2d* for details and output shape.

Parameters

- **output_size** – the target output size (single integer or double-integer tuple)
- **return_indices** – whether to return pooling indices. Default: False

17.2.14 adaptive_max_pool3d

torch.nn.functional.**adaptive_max_pool3d**(**args*, ***kwargs*)
Applies a 3D adaptive max pooling over an input signal composed of several input planes.

See *AdaptiveMaxPool3d* for details and output shape.

Parameters

- **output_size** – the target output size (single integer or triple-integer tuple)
- **return_indices** – whether to return pooling indices. Default: False

17.2.15 adaptive_avg_pool1d

torch.nn.functional.**adaptive_avg_pool1d**(*input*, *output_size*) → Tensor
Applies a 1D adaptive average pooling over an input signal composed of several input planes.

See *AdaptiveAvgPool1d* for details and output shape.

Parameters **output_size** – the target output size (single integer)

17.2.16 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.

Parameters **output_size** – the target output size (single integer or double-integer tuple)

17.2.17 adaptive_avg_pool3d

torch.nn.functional.**adaptive_avg_pool3d**(*input*, *output_size*)
Applies a 3D adaptive average pooling over an input signal composed of several input planes.

See *AdaptiveAvgPool3d* for details and output shape.

Parameters **output_size** – the target output size (single integer or triple-integer tuple)

17.3 Non-linear activation functions

17.3.1 threshold

torch.nn.functional.**threshold**(*input*, *threshold*, *value*, *inplace=False*)
Thresholds each element of the input Tensor.

See *Threshold* for more details.

torch.nn.functional.**threshold_**(*input*, *threshold*, *value*) → Tensor
In-place version of *threshold()*.

17.3.2 relu

torch.nn.functional.**relu**(*input*, *inplace=False*) → Tensor
Applies the rectified linear unit function element-wise. See *ReLU* for more details.

torch.nn.functional.**relu_**(*input*) → Tensor
In-place version of *relu()*.

17.3.3 hardtanh

torch.nn.functional.**hardtanh**(*input*, *min_val=-1.*, *max_val=1.*, *inplace=False*) → Tensor
Applies the HardTanh function element-wise. See *Hardtanh* for more details.

torch.nn.functional.**hardtanh_**(*input*, *min_val=-1.*, *max_val=1.*) → Tensor
In-place version of *hardtanh()*.

17.3.4 relu6

torch.nn.functional.**relu6**(*input*, *inplace=False*) → Tensor
Applies the element-wise function $\text{ReLU6}(x) = \min(\max(0, x), 6)$.

See *ReLU6* for more details.

17.3.5 elu

torch.nn.functional.**elu**(*input*, *alpha=1.0*, *inplace=False*)
Applies element-wise, $\text{ELU}(x) = \max(0, x) + \min(0, \alpha * (\exp(x) - 1))$.

See *ELU* for more details.

torch.nn.functional.**elu_**(*input*, *alpha=1.*) → Tensor
In-place version of *elu()*.

17.3.6 selu

torch.nn.functional.**selu**(*input*, *inplace=False*) → Tensor
Applies element-wise, SELU$(x) = scale * (\max(0, x) + \min(0, \alpha * (\exp(x) - 1)))$, with $\alpha = 1.6732632423543772848170429916717$ and $scale = 1.0507009873554804934193349852946$.

See *SELU* for more details.

17.3.7 celu

torch.nn.functional.**celu**(*input*, *alpha=1.*, *inplace=False*) → Tensor
Applies element-wise, CELU$(x) = \max(0, x) + \min(0, \alpha * (\exp(x/\alpha) - 1))$.

See *CELU* for more details.

17.3.8 leaky_relu

torch.nn.functional.**leaky_relu**(*input*, *negative_slope=0.01*, *inplace=False*) → Tensor
Applies element-wise, LeakyReLU$(x) = \max(0, x) + \text{negative_slope} * \min(0, x)$

See *LeakyReLU* for more details.

torch.nn.functional.**leaky_relu_**(*input*, *negative_slope=0.01*) → Tensor
In-place version of *leaky_relu()*.

17.3.9 prelu

torch.nn.functional.**prelu**(*input*, *weight*) → Tensor
Applies element-wise the function PReLU$(x) = \max(0, x) + \text{weight} * \min(0, x)$ where weight is a learnable parameter.

See *PReLU* for more details.

17.3.10 rrelu

torch.nn.functional.**rrelu**(*input*, *lower=1./8*, *upper=1./3*, *training=False*, *inplace=False*) → Tensor
Randomized leaky ReLU.

See *RReLU* for more details.

torch.nn.functional.**rrelu_**(*input*, *lower=1./8*, *upper=1./3*, *training=False*) → Tensor
In-place version of *rrelu()*.

17.3.11 glu

torch.nn.functional.**glu**(*input*, *dim=-1*) → Tensor
The gated linear unit. Computes:

$$\text{GLU}(a, b) = a \otimes \sigma(b)$$

where *input* is split in half along *dim* to form a and b, σ is the sigmoid function and $\otimes$ is the element-wise product between matrices.

See Language Modeling with Gated Convolutional Networks.

Parameters

- **input** (Tensor) – input tensor
- **dim** (*int*) – dimension on which to split the input. Default: -1

17.3.12 gelu

17.3.13 logsigmoid

torch.nn.functional.**logsigmoid**(*input*) → Tensor

Applies element-wise LogSigmoid$(x_i) = \log\left(\frac{1}{1+\exp(-x_i)}\right)$

See *LogSigmoid* for more details.

17.3.14 hardshrink

torch.nn.functional.**hardshrink**(*input*, *lambd=0.5*) → Tensor

Applies the hard shrinkage function element-wise

See *Hardshrink* for more details.

17.3.15 tanhshrink

torch.nn.functional.**tanhshrink**(*input*) → Tensor

Applies element-wise, Tanhshrink$(x) = x - \text{Tanh}(x)$

See *Tanhshrink* for more details.

17.3.16 softsign

torch.nn.functional.**softsign**(*input*) → Tensor

Applies element-wise, the function SoftSign$(x) = \frac{x}{1+|x|}$

See *Softsign* for more details.

17.3.17 softplus

torch.nn.functional.**softplus**(*input*, *beta=1*, *threshold=20*) → Tensor

17.3.18 softmin

torch.nn.functional.**softmin**(*input*, *dim=None*, *_stacklevel=3*, *dtype=None*)

Applies a softmin function.

Note that Softmin$(x) =$ Softmax$(-x)$. See softmax definition for mathematical formula.

See *Softmin* for more details.

Parameters

- **input** (Tensor) – input

- **dim** (*int*) – A dimension along which softmin will be computed (so every slice along dim will sum to 1).
- **dtype** (torch.dtype, optional) – the desired data type of returned tensor. If specified, the input tensor is casted to dtype before the operation is performed. This is useful for preventing data type overflows. Default: None.

17.3.19 softmax

torch.nn.functional.**softmax**(*input*, *dim=None*, *_stacklevel=3*, *dtype=None*)

Applies a softmax function.

Softmax is defined as:

$\text{Softmax}(x_i) = \frac{exp(x_i)}{\sum_j exp(x_j)}$

It is applied to all slices along dim, and will re-scale them so that the elements lie in the range *[0, 1]* and sum to 1.

See *Softmax* for more details.

Parameters

- **input** (Tensor) – input
- **dim** (*int*) – A dimension along which softmax will be computed.
- **dtype** (torch.dtype, optional) – the desired data type of returned tensor. If specified, the input tensor is casted to dtype before the operation is performed. This is useful for preventing data type overflows. Default: None.

Note: This function doesnt work directly with NLLLoss, which expects the Log to be computed between the Softmax and itself. Use log_softmax instead (its faster and has better numerical properties).

17.3.20 softshrink

torch.nn.functional.**softshrink**(*input*, *lambd=0.5*) → Tensor

Applies the soft shrinkage function elementwise

See *Softshrink* for more details.

17.3.21 gumbel_softmax

torch.nn.functional.**gumbel_softmax**(*logits*, *tau=1*, *hard=False*, *eps=1e-10*, *dim=-1*)

Samples from the Gumbel-Softmax distribution and optionally discretizes.

Parameters

- **logits** – *[, num_features]* unnormalized log probabilities
- **tau** – non-negative scalar temperature
- **hard** – if True, the returned samples will be discretized as one-hot vectors, but will be differentiated as if it is the soft sample in autograd
- **dim** (*int*) – A dimension along which softmax will be computed. Default: -1.

Returns Sampled tensor of same shape as *logits* from the Gumbel-Softmax distribution. If `hard=True`, the returned samples will be one-hot, otherwise they will be probability distributions that sum to 1 across *dim*.

Note: This function is here for legacy reasons, may be removed from nn.Functional in the future.

Note: The main trick for *hard* is to do *y_hard - y_soft.detach() + y_soft*

It achieves two things: - makes the output value exactly one-hot (since we add then subtract y_soft value) - makes the gradient equal to y_soft gradient (since we strip all other gradients)

Examples::

```
>>> logits = torch.randn(20, 32)
>>> # Sample soft categorical using reparametrization trick:
>>> F.gumbel_softmax(logits, tau=1, hard=False)
>>> # Sample hard categorical using "Straight-through" trick:
>>> F.gumbel_softmax(logits, tau=1, hard=True)
```

17.3.22 log_softmax

`torch.nn.functional.`**`log_softmax`**(*input*, *dim=None*, *_stacklevel=3*, *dtype=None*)

Applies a softmax followed by a logarithm.

While mathematically equivalent to log(softmax(x)), doing these two operations separately is slower, and numerically unstable. This function uses an alternative formulation to compute the output and gradient correctly.

See *LogSoftmax* for more details.

Parameters

- **input** (Tensor) – input
- **dim** (*int*) – A dimension along which log_softmax will be computed.
- **dtype** (`torch.dtype`, optional) – the desired data type of returned tensor. If specified, the input tensor is casted to `dtype` before the operation is performed. This is useful for preventing data type overflows. Default: None.

17.3.23 tanh

`torch.nn.functional.`**`tanh`**(*input*) → Tensor

Applies element-wise, $\text{Tanh}(x) = \tanh(x) = \frac{\exp(x)-\exp(-x)}{\exp(x)+\exp(-x)}$

See *Tanh* for more details.

17.3.24 sigmoid

`torch.nn.functional.`**`sigmoid`**(*input*) → Tensor

Applies the element-wise function $\text{Sigmoid}(x) = \frac{1}{1+\exp(-x)}$

See *Sigmoid* for more details.

17.4 Normalization functions

17.4.1 batch_norm

torch.nn.functional.**batch_norm**(*input*, *running_mean*, *running_var*, *weight=None*, *bias=None*, *training=False*, *momentum=0.1*, *eps=1e-05*)

Applies Batch Normalization for each channel across a batch of data.

See *BatchNorm1d*, *BatchNorm2d*, *BatchNorm3d* for details.

17.4.2 instance_norm

torch.nn.functional.**instance_norm**(*input*, *running_mean=None*, *running_var=None*, *weight=None*, *bias=None*, *use_input_stats=True*, *momentum=0.1*, *eps=1e-05*)

Applies Instance Normalization for each channel in each data sample in a batch.

See *InstanceNorm1d*, *InstanceNorm2d*, *InstanceNorm3d* for details.

17.4.3 layer_norm

torch.nn.functional.**layer_norm**(*input*, *normalized_shape*, *weight=None*, *bias=None*, *eps=1e-05*)

Applies Layer Normalization for last certain number of dimensions.

See *LayerNorm* for details.

17.4.4 local_response_norm

torch.nn.functional.**local_response_norm**(*input*, *size*, *alpha=0.0001*, *beta=0.75*, *k=1.0*)

Applies local response normalization over an input signal composed of several input planes, where channels occupy the second dimension. Applies normalization across channels.

See *LocalResponseNorm* for details.

17.4.5 normalize

torch.nn.functional.**normalize**(*input*, *p=2*, *dim=1*, *eps=1e-12*, *out=None*)

Performs L_p normalization of inputs over specified dimension.

For a tensor `input` of sizes $(n_0, ..., n_{dim}, ..., n_k)$, each n_{dim} -element vector v along dimension `dim` is transformed as

$$v = \frac{v}{\max(\|v\|_p, \epsilon)}.$$

With the default arguments it uses the Euclidean norm over vectors along dimension 1 for normalization.

Parameters

- **input** – input tensor of any shape
- **p** (*float*) – the exponent value in the norm formulation. Default: 2
- **dim** (*int*) – the dimension to reduce. Default: 1

- **eps** (*float*) – small value to avoid division by zero. Default: 1e-12
- **out** (*Tensor, optional*) – the output tensor. If `out` is used, this operation wont be differentiable.

17.5 Linear functions

17.5.1 linear

`torch.nn.functional.`**`linear`**(*input*, *weight*, *bias=None*)

Applies a linear transformation to the incoming data: $y = xA^T + b$.

Shape:

- Input: $(N, *, in\_features)$ where * means any number of additional dimensions
- Weight: $(out\_features, in\_features)$
- Bias: $(out\_features)$
- Output: $(N, *, out\_features)$

17.5.2 bilinear

`torch.nn.functional.`**`bilinear`**(*input1*, *input2*, *weight*, *bias=None*)

17.6 Dropout functions

17.6.1 dropout

`torch.nn.functional.`**`dropout`**(*input*, *p=0.5*, *training=True*, *inplace=False*)

During training, randomly zeroes some of the elements of the input tensor with probability `p` using samples from a Bernoulli distribution.

See *Dropout* for details.

Parameters

- **p** – probability of an element to be zeroed. Default: 0.5
- **training** – apply dropout if is `True`. Default: `True`
- **inplace** – If set to `True`, will do this operation in-place. Default: `False`

17.6.2 alpha_dropout

`torch.nn.functional.`**`alpha_dropout`**(*input*, *p=0.5*, *training=False*, *inplace=False*)

Applies alpha dropout to the input.

See *AlphaDropout* for details.

17.6.3 dropout2d

torch.nn.functional.**dropout2d**(*input*, *p=0.5*, *training=True*, *inplace=False*)

Randomly zero out entire channels (a channel is a 2D feature map, e.g., the j-th channel of the i-th sample in the batched input is a 2D tensor input$[i, j]$) of the input tensor). Each channel will be zeroed out independently on every forward call with probability `p` using samples from a Bernoulli distribution.

See *Dropout2d* for details.

Parameters

- **p** – probability of a channel to be zeroed. Default: 0.5
- **training** – apply dropout if is `True`. Default: `True`
- **inplace** – If set to `True`, will do this operation in-place. Default: `False`

17.6.4 dropout3d

torch.nn.functional.**dropout3d**(*input*, *p=0.5*, *training=True*, *inplace=False*)

Randomly zero out entire channels (a channel is a 3D feature map, e.g., the j-th channel of the i-th sample in the batched input is a 3D tensor input$[i, j]$) of the input tensor). Each channel will be zeroed out independently on every forward call with probability `p` using samples from a Bernoulli distribution.

See *Dropout3d* for details.

Parameters

- **p** – probability of a channel to be zeroed. Default: 0.5
- **training** – apply dropout if is `True`. Default: `True`
- **inplace** – If set to `True`, will do this operation in-place. Default: `False`

17.7 Sparse functions

17.7.1 embedding

torch.nn.functional.**embedding**(*input*, *weight*, *padding_idx=None*, *max_norm=None*, *norm_type=2.0*, *scale_grad_by_freq=False*, *sparse=False*)

A simple lookup table that looks up embeddings in a fixed dictionary and size.

This module is often used to retrieve word embeddings using indices. The input to the module is a list of indices, and the embedding matrix, and the output is the corresponding word embeddings.

See *torch.nn.Embedding* for more details.

Parameters

- **input** (*LongTensor*) – Tensor containing indices into the embedding matrix
- **weight** (*Tensor*) – The embedding matrix with number of rows equal to the maximum possible index + 1, and number of columns equal to the embedding size
- **padding_idx** (*int, optional*) – If given, pads the output with the embedding vector at `padding_idx` (initialized to zeros) whenever it encounters the index.
- **max_norm** (*float, optional*) – If given, each embedding vector with norm larger than `max_norm` is renormalized to have norm `max_norm`. Note: this will modify `weight` in-place.

- **norm_type** (*float, optional*) – The p of the p-norm to compute for the `max_norm` option. Default `2`.
- **scale_grad_by_freq** (*boolean, optional*) – If given, this will scale gradients by the inverse of frequency of the words in the mini-batch. Default `False`.
- **sparse** (*bool, optional*) – If `True`, gradient w.r.t. `weight` will be a sparse tensor. See Notes under *torch.nn.Embedding* for more details regarding sparse gradients.

Shape:

- Input: LongTensor of arbitrary shape containing the indices to extract
- **Weight: Embedding matrix of floating point type with shape *(V, embedding_dim)*,** where V = maximum index + 1 and embedding_dim = the embedding size
- Output: *(*, embedding_dim)*, where * is the input shape

Examples:

```
>>> # a batch of 2 samples of 4 indices each
>>> input = torch.tensor([[1,2,4,5],[4,3,2,9]])
>>> # an embedding matrix containing 10 tensors of size 3
>>> embedding_matrix = torch.rand(10, 3)
>>> F.embedding(input, embedding_matrix)
tensor([[[ 0.8490,  0.9625,  0.6753],
         [ 0.9666,  0.7761,  0.6108],
         [ 0.6246,  0.9751,  0.3618],
         [ 0.4161,  0.2419,  0.7383]],

        [[ 0.6246,  0.9751,  0.3618],
         [ 0.0237,  0.7794,  0.0528],
         [ 0.9666,  0.7761,  0.6108],
         [ 0.3385,  0.8612,  0.1867]]])

>>> # example with padding_idx
>>> weights = torch.rand(10, 3)
>>> weights[0, :].zero_()
>>> embedding_matrix = weights
>>> input = torch.tensor([[0,2,0,5]])
>>> F.embedding(input, embedding_matrix, padding_idx=0)
tensor([[[ 0.0000,  0.0000,  0.0000],
         [ 0.5609,  0.5384,  0.8720],
         [ 0.0000,  0.0000,  0.0000],
         [ 0.6262,  0.2438,  0.7471]]])
```

17.7.2 embedding_bag

`torch.nn.functional.`**embedding_bag**(*input, weight, offsets=None, max_norm=None, norm_type=2, scale_grad_by_freq=False, mode='mean', sparse=False*)

Computes sums, means or maxes of *bags* of embeddings, without instantiating the intermediate embeddings.

See *torch.nn.EmbeddingBag* for more details.

Note: When using the CUDA backend, this operation may induce nondeterministic behaviour in its backward pass that is not easily switched off. Please see the notes on *Reproducibility* for background.

Parameters

- **input** (*LongTensor*) – Tensor containing bags of indices into the embedding matrix
- **weight** (Tensor) – The embedding matrix with number of rows equal to the maximum possible index + 1, and number of columns equal to the embedding size
- **offsets** (*LongTensor, optional*) – Only used when `input` is 1D. `offsets` determines the starting index position of each bag (sequence) in `input`.
- **max_norm** (*float, optional*) – If given, each embedding vector with norm larger than `max_norm` is renormalized to have norm `max_norm`. Note: this will modify `weight` in-place.
- **norm_type** (*float, optional*) – The `p` in the `p`-norm to compute for the `max_norm` option. Default `2`.
- **scale_grad_by_freq** (*boolean, optional*) – if given, this will scale gradients by the inverse of frequency of the words in the mini-batch. Default `False`. Note: this option is not supported when `mode="max"`.
- **mode** (*string, optional*) – `"sum"`, `"mean"` or `"max"`. Specifies the way to reduce the bag. Default: `"mean"`
- **sparse** (*bool, optional*) – if `True`, gradient w.r.t. `weight` will be a sparse tensor. See Notes under *torch.nn.Embedding* for more details regarding sparse gradients. Note: this option is not supported when `mode="max"`.

Shape:

- `input` (LongTensor) and `offsets` (LongTensor, optional)
  - If `input` is 2D of shape *(B, N)*,

    it will be treated as `B` bags (sequences) each of fixed length `N`, and this will return `B` values aggregated in a way depending on the `mode`. `offsets` is ignored and required to be `None` in this case.
  - If `input` is 1D of shape *(N)*,

    it will be treated as a concatenation of multiple bags (sequences). `offsets` is required to be a 1D tensor containing the starting index positions of each bag in `input`. Therefore, for `offsets` of shape *(B)*, `input` will be viewed as having `B` bags. Empty bags (i.e., having 0-length) will have returned vectors filled by zeros.
- `weight` (Tensor): the learnable weights of the module of shape *(num_embeddings, embedding_dim)*
- `output`: aggregated embedding values of shape *(B, embedding_dim)*

Examples:

```
>>> # an Embedding module containing 10 tensors of size 3
>>> embedding_matrix = torch.rand(10, 3)
>>> # a batch of 2 samples of 4 indices each
>>> input = torch.tensor([1,2,4,5,4,3,2,9])
>>> offsets = torch.tensor([0,4])
>>> F.embedding_bag(embedding_matrix, input, offsets)
tensor([[ 0.3397,  0.3552,  0.5545],
        [ 0.5893,  0.4386,  0.5882]])
```

17.7.3 one_hot

torch.nn.functional.**one_hot**(*tensor*, *num_classes=0*) → LongTensor

Takes LongTensor with index values of shape `(*)` and returns a tensor of shape `(*, num_classes)` that have zeros everywhere except where the index of last dimension matches the corresponding value of the input tensor, in which case it will be 1.

See also One-hot on Wikipedia .

Parameters

- **tensor** (*LongTensor*) – class values of any shape.
- **num_classes** (*int*) – Total number of classes. If set to -1, the number of classes will be inferred as one greater than the largest class value in the input tensor.

Returns LongTensor that has one more dimension with 1 values at the index of last dimension indicated by the input, and 0 everywhere else.

Examples

```
>>> F.one_hot(torch.arange(0, 5) % 3)
tensor([[1, 0, 0],
        [0, 1, 0],
        [0, 0, 1],
        [1, 0, 0],
        [0, 1, 0]])
>>> F.one_hot(torch.arange(0, 5) % 3, num_classes=5)
tensor([[1, 0, 0, 0, 0],
        [0, 1, 0, 0, 0],
        [0, 0, 1, 0, 0],
        [1, 0, 0, 0, 0],
        [0, 1, 0, 0, 0]])
>>> F.one_hot(torch.arange(0, 6).view(3,2) % 3)
tensor([[[1, 0, 0],
         [0, 1, 0]],
        [[0, 0, 1],
         [1, 0, 0]],
        [[0, 1, 0],
         [0, 0, 1]]])
```

17.8 Distance functions

17.8.1 pairwise_distance

torch.nn.functional.**pairwise_distance**(*x1*, *x2*, *p=2.0*, *eps=1e-06*, *keepdim=False*)

See *torch.nn.PairwiseDistance* for details

17.8.2 cosine_similarity

torch.nn.functional.**cosine_similarity**(*x1*, *x2*, *dim=1*, *eps=1e-8*) → Tensor

Returns cosine similarity between x1 and x2, computed along dim.

$$\text{similarity} = \frac{x_1 \cdot x_2}{\max(\|x_1\|_2 \cdot \|x_2\|_2, \epsilon)}$$

Parameters

- **x1** (Tensor) – First input.
- **x2** (Tensor) – Second input (of size matching x1).
- **dim** (*int, optional*) – Dimension of vectors. Default: 1
- **eps** (*float, optional*) – Small value to avoid division by zero. Default: 1e-8

Shape:

- Input: $(*_1, D, *_2)$ where D is at position *dim*.
- Output: $(*_1, *_2)$ where 1 is at position *dim*.

Example:

```
>>> input1 = torch.randn(100, 128)
>>> input2 = torch.randn(100, 128)
>>> output = F.cosine_similarity(input1, input2)
>>> print(output)
```

17.8.3 pdist

torch.nn.functional.**pdist**(*input*, *p=2*) → Tensor

Computes the p-norm distance between every pair of row vectors in the input. This is identical to the upper triangular portion, excluding the diagonal, of *torch.norm(input[:, None] - input, dim=2, p=p)*. This function will be faster if the rows are contiguous.

If input has shape $N \times M$ then the output will have shape $\frac{1}{2}N(N-1)$.

This function is equivalent to *scipy.spatial.distance.pdist(input, minkowski, p=p)* if $p \in (0, \infty)$. When $p = 0$ it is equivalent to *scipy.spatial.distance.pdist(input, hamming) * M*. When $p = \infty$, the closest scipy function is *scipy.spatial.distance.pdist(xn, lambda x, y: np.abs(x - y).max())*.

Parameters

- **input** – input tensor of shape $N \times M$.
- **p** – p value for the p-norm distance to calculate between each vector pair $\in [0, \infty]$.

17.9 Loss functions

17.9.1 binary_cross_entropy

torch.nn.functional.**binary_cross_entropy**(*input*, *target*, *weight=None*, *size_average=None*, *reduce=None*, *reduction='mean'*)

Function that measures the Binary Cross Entropy between the target and the output.

See *BCELoss* for details.

Parameters

- **input** – Tensor of arbitrary shape
- **target** – Tensor of the same shape as input

- **weight** (*Tensor, optional*) – a manual rescaling weight if provided its repeated to match input tensor shape
- **size_average** (*bool, optional*) – Deprecated (see `reduction`). By default, the losses are averaged over each loss element in the batch. Note that for some losses, there multiple elements per sample. If the field `size_average` is set to `False`, the losses are instead summed for each minibatch. Ignored when reduce is `False`. Default: `True`
- **reduce** (*bool, optional*) – Deprecated (see `reduction`). By default, the losses are averaged or summed over observations for each minibatch depending on `size_average`. When `reduce` is `False`, returns a loss per batch element instead and ignores `size_average`. Default: `True`
- **reduction** (*string, optional*) – Specifies the reduction to apply to the output: `'none' | 'mean' | 'sum'`. `'none'`: no reduction will be applied, `'mean'`: the sum of the output will be divided by the number of elements in the output, `'sum'`: the output will be summed. Note: `size_average` and `reduce` are in the process of being deprecated, and in the meantime, specifying either of those two args will override `reduction`. Default: `'mean'`

Examples:

```
>>> input = torch.randn((3, 2), requires_grad=True)
>>> target = torch.rand((3, 2), requires_grad=False)
>>> loss = F.binary_cross_entropy(F.sigmoid(input), target)
>>> loss.backward()
```

17.9.2 binary_cross_entropy_with_logits

torch.nn.functional.**binary_cross_entropy_with_logits**(*input*, *target*, *weight=None*, *size_average=None*, *reduce=None*, *reduction='mean'*, *pos_weight=None*)

Function that measures Binary Cross Entropy between target and output logits.

See *BCEWithLogitsLoss* for details.

Parameters

- **input** – Tensor of arbitrary shape
- **target** – Tensor of the same shape as input
- **weight** (*Tensor, optional*) – a manual rescaling weight if provided its repeated to match input tensor shape
- **size_average** (*bool, optional*) – Deprecated (see `reduction`). By default, the losses are averaged over each loss element in the batch. Note that for some losses, there multiple elements per sample. If the field `size_average` is set to `False`, the losses are instead summed for each minibatch. Ignored when reduce is `False`. Default: `True`
- **reduce** (*bool, optional*) – Deprecated (see `reduction`). By default, the losses are averaged or summed over observations for each minibatch depending on `size_average`. When `reduce` is `False`, returns a loss per batch element instead and ignores `size_average`. Default: `True`
- **reduction** (*string, optional*) – Specifies the reduction to apply to the output: `'none' | 'mean' | 'sum'`. `'none'`: no reduction will be applied, `'mean'`: the sum

of the output will be divided by the number of elements in the output, `'sum'`: the output will be summed. Note: `size_average` and `reduce` are in the process of being deprecated, and in the meantime, specifying either of those two args will override `reduction`. Default: `'mean'`

- **pos_weight** (*Tensor, optional*) – a weight of positive examples. Must be a vector with length equal to the number of classes.

Examples:

```
>>> input = torch.randn(3, requires_grad=True)
>>> target = torch.empty(3).random_(2)
>>> loss = F.binary_cross_entropy_with_logits(input, target)
>>> loss.backward()
```

17.9.3 poisson_nll_loss

torch.nn.functional.**poisson_nll_loss**(*input*, *target*, *log_input=True*, *full=False*, *size_average=None*, *eps=1e-08*, *reduce=None*, *reduction='mean'*)

Poisson negative log likelihood loss.

See *PoissonNLLLoss* for details.

Parameters

- **input** – expectation of underlying Poisson distribution.
- **target** – random sample $target \sim \text{Poisson}(input)$.
- **log_input** – if `True` the loss is computed as $\exp(\text{input}) - \text{target} * \text{input}$, if `False` then loss is $\text{input} - \text{target} * \log(\text{input} + \text{eps})$. Default: `True`
- **full** – whether to compute full loss, i. e. to add the Stirling approximation term. Default: `False` $\text{target} * \log(\text{target}) - \text{target} + 0.5 * \log(2 * \pi * \text{target})$.
- **size_average** (*bool, optional*) – Deprecated (see `reduction`). By default, the losses are averaged over each loss element in the batch. Note that for some losses, there multiple elements per sample. If the field `size_average` is set to `False`, the losses are instead summed for each minibatch. Ignored when reduce is `False`. Default: `True`
- **eps** (*float, optional*) – Small value to avoid evaluation of $\log(0)$ when log_input`=``False`. Default: 1e-8
- **reduce** (*bool, optional*) – Deprecated (see `reduction`). By default, the losses are averaged or summed over observations for each minibatch depending on `size_average`. When `reduce` is `False`, returns a loss per batch element instead and ignores `size_average`. Default: `True`
- **reduction** (*string, optional*) – Specifies the reduction to apply to the output: `'none'` | `'mean'` | `'sum'`. `'none'`: no reduction will be applied, `'mean'`: the sum of the output will be divided by the number of elements in the output, `'sum'`: the output will be summed. Note: `size_average` and `reduce` are in the process of being deprecated, and in the meantime, specifying either of those two args will override `reduction`. Default: `'mean'`

17.9.4 cosine_embedding_loss

torch.nn.functional.**cosine_embedding_loss**(*input1*, *input2*, *target*, *margin=0*, *size_average=None*, *reduce=None*, *reduction='mean'*) → Tensor

See *CosineEmbeddingLoss* for details.

17.9.5 cross_entropy

torch.nn.functional.**cross_entropy**(*input*, *target*, *weight=None*, *size_average=None*, *ignore_index=-100*, *reduce=None*, *reduction='mean'*)

This criterion combines *log_softmax* and *nll_loss* in a single function.

See *CrossEntropyLoss* for details.

Parameters

- **input** (Tensor) – (N, C) where C = *number of classes* or (N, C, H, W) in case of 2D Loss, or $(N, C, d_1, d_2, ..., d_K)$ where $K \geq 1$ in the case of K-dimensional loss.
- **target** (Tensor) – (N) where each value is $0 \leq \text{targets}[i] \leq C-1$, or $(N, d_1, d_2, ..., d_K)$ where $K \geq 1$ for K-dimensional loss.
- **weight** (Tensor, *optional*) – a manual rescaling weight given to each class. If given, has to be a Tensor of size C
- **size_average** (*bool*, *optional*) – Deprecated (see `reduction`). By default, the losses are averaged over each loss element in the batch. Note that for some losses, there multiple elements per sample. If the field `size_average` is set to `False`, the losses are instead summed for each minibatch. Ignored when reduce is `False`. Default: `True`
- **ignore_index** (*int*, *optional*) – Specifies a target value that is ignored and does not contribute to the input gradient. When `size_average` is `True`, the loss is averaged over non-ignored targets. Default: -100
- **reduce** (*bool*, *optional*) – Deprecated (see `reduction`). By default, the losses are averaged or summed over observations for each minibatch depending on `size_average`. When `reduce` is `False`, returns a loss per batch element instead and ignores `size_average`. Default: `True`
- **reduction** (*string*, *optional*) – Specifies the reduction to apply to the output: `'none'` | `'mean'` | `'sum'`. `'none'`: no reduction will be applied, `'mean'`: the sum of the output will be divided by the number of elements in the output, `'sum'`: the output will be summed. Note: `size_average` and `reduce` are in the process of being deprecated, and in the meantime, specifying either of those two args will override `reduction`. Default: `'mean'`

Examples:

```
>>> input = torch.randn(3, 5, requires_grad=True)
>>> target = torch.randint(5, (3,), dtype=torch.int64)
>>> loss = F.cross_entropy(input, target)
>>> loss.backward()
```

17.9.6 ctc_loss

torch.nn.functional.**ctc_loss**(*log_probs*, *targets*, *input_lengths*, *target_lengths*, *blank=0*, *reduction='mean'*, *zero_infinity=False*)

The Connectionist Temporal Classification loss.

See *CTCLoss* for details.

Note: In some circumstances when using the CUDA backend with CuDNN, this operator may select a nondeterministic algorithm to increase performance. If this is undesirable, you can try to make the operation deterministic (potentially at a performance cost) by setting `torch.backends.cudnn.deterministic = True`. Please see the notes on *Reproducibility* for background.

Note: When using the CUDA backend, this operation may induce nondeterministic behaviour in its backward pass that is not easily switched off. Please see the notes on *Reproducibility* for background.

Parameters

- **log_probs** – (T, N, C) where C = *number of characters in alphabet including blank*, T = *input length*, and N = *batch size*. The logarithmized probabilities of the outputs (e.g. obtained with *torch.nn.functional.log_softmax()*).
- **targets** – (N, S) or *(sum(target_lengths))*. Targets cannot be blank. In the second form, the targets are assumed to be concatenated.
- **input_lengths** – (N). Lengths of the inputs (must each be $\leq T$)
- **target_lengths** – (N). Lengths of the targets
- **blank** (*int, optional*) – Blank label. Default 0.
- **reduction** (*string, optional*) – Specifies the reduction to apply to the output: `'none'` | `'mean'` | `'sum'`. `'none'`: no reduction will be applied, `'mean'`: the output losses will be divided by the target lengths and then the mean over the batch is taken, `'sum'`: the output will be summed. Default: `'mean'`
- **zero_infinity** (*bool, optional*) – Whether to zero infinite losses and the associated gradients. Default: `False` Infinite losses mainly occur when the inputs are too short to be aligned to the targets.

Example:

```
>>> log_probs = torch.randn(50, 16, 20).log_softmax(2).detach().requires_grad_()
>>> targets = torch.randint(1, 20, (16, 30), dtype=torch.long)
>>> input_lengths = torch.full((16,), 50, dtype=torch.long)
>>> target_lengths = torch.randint(10,30,(16,), dtype=torch.long)
>>> loss = F.ctc_loss(log_probs, targets, input_lengths, target_lengths)
>>> loss.backward()
```

17.9.7 hinge_embedding_loss

torch.nn.functional.**hinge_embedding_loss**(*input*, *target*, *margin=1.0*, *size_average=None*, *reduce=None*, *reduction='mean'*) → Tensor

See *HingeEmbeddingLoss* for details.

17.9.8 kl_div

torch.nn.functional.**kl_div**(*input*, *target*, *size_average=None*, *reduce=None*, *reduction='mean'*)

The **'Kullback-Leibler divergence'_** Loss.

See *KLDivLoss* for details.

Parameters

- **input** – Tensor of arbitrary shape
- **target** – Tensor of the same shape as input
- **size_average** (*bool, optional*) – Deprecated (see reduction). By default, the losses are averaged over each loss element in the batch. Note that for some losses, there multiple elements per sample. If the field size_average is set to False, the losses are instead summed for each minibatch. Ignored when reduce is False. Default: True
- **reduce** (*bool, optional*) – Deprecated (see reduction). By default, the losses are averaged or summed over observations for each minibatch depending on size_average. When reduce is False, returns a loss per batch element instead and ignores size_average. Default: True
- **reduction** (*string, optional*) – Specifies the reduction to apply to the output: 'none' | 'batchmean' | 'sum' | 'mean'. 'none': no reduction will be applied 'batchmean': the sum of the output will be divided by the batchsize 'sum': the output will be summed 'mean': the output will be divided by the number of elements in the output Default: 'mean'

Note: size_average and reduce are in the process of being deprecated, and in the meantime, specifying either of those two args will override reduction.

Note: :attr:reduction = 'mean' doesnt return the true kl divergence value, please use :attr:reduction = 'batchmean' which aligns with KL math definition. In the next major release, 'mean' will be changed to be the same as batchmean.

17.9.9 l1_loss

torch.nn.functional.**l1_loss**(*input*, *target*, *size_average=None*, *reduce=None*, *reduction='mean'*) → Tensor

Function that takes the mean element-wise absolute value difference.

See *L1Loss* for details.

17.9.10 mse_loss

torch.nn.functional.**mse_loss**(*input*, *target*, *size_average=None*, *reduce=None*, *reduction='mean'*) → Tensor

Measures the element-wise mean squared error.

See *MSELoss* for details.

17.9.11 margin_ranking_loss

torch.nn.functional.**margin_ranking_loss**(*input1*, *input2*, *target*, *margin=0*, *size_average=None*, *reduce=None*, *reduction='mean'*) → Tensor

See *MarginRankingLoss* for details.

17.9.12 multilabel_margin_loss

torch.nn.functional.**multilabel_margin_loss**(*input*, *target*, *size_average=None*, *reduce=None*, *reduction='mean'*) → Tensor

See *MultiLabelMarginLoss* for details.

17.9.13 multilabel_soft_margin_loss

torch.nn.functional.**multilabel_soft_margin_loss**(*input*, *target*, *weight=None*, *size_average=None*) → Tensor

See *MultiLabelSoftMarginLoss* for details.

17.9.14 multi_margin_loss

torch.nn.functional.**multi_margin_loss**(*input*, *target*, *p=1*, *margin=1.0*, *weight=None*, *size_average=None*, *reduce=None*, *reduction='mean'*)

multi_margin_loss(input, target, p=1, margin=1, weight=None, size_average=None, reduce=None, reduction=mean) -> Tensor

See *MultiMarginLoss* for details.

17.9.15 nll_loss

torch.nn.functional.**nll_loss**(*input*, *target*, *weight=None*, *size_average=None*, *ignore_index=-100*, *reduce=None*, *reduction='mean'*)

The negative log likelihood loss.

See *NLLLoss* for details.

Parameters

- **input** – (N, C) where C = *number of classes* or (N, C, H, W) in case of 2D Loss, or $(N, C, d_1, d_2, ..., d_K)$ where $K \geq 1$ in the case of K-dimensional loss.
- **target** – (N) where each value is $0 \leq \text{targets}[i] \leq C - 1$, or $(N, d_1, d_2, ..., d_K)$ where $K \geq 1$ for K-dimensional loss.
- **weight** (*Tensor, optional*) – a manual rescaling weight given to each class. If given, has to be a Tensor of size C
- **size_average** (*bool, optional*) – Deprecated (see `reduction`). By default, the losses are averaged over each loss element in the batch. Note that for some losses, there multiple elements per sample. If the field `size_average` is set to `False`, the losses are instead summed for each minibatch. Ignored when reduce is `False`. Default: `True`

- **ignore_index** (*int, optional*) – Specifies a target value that is ignored and does not contribute to the input gradient. When `size_average` is `True`, the loss is averaged over non-ignored targets. Default: -100
- **reduce** (*bool, optional*) – Deprecated (see `reduction`). By default, the losses are averaged or summed over observations for each minibatch depending on `size_average`. When `reduce` is `False`, returns a loss per batch element instead and ignores `size_average`. Default: `True`
- **reduction** (*string, optional*) – Specifies the reduction to apply to the output: `'none'` | `'mean'` | `'sum'`. `'none'`: no reduction will be applied, `'mean'`: the sum of the output will be divided by the number of elements in the output, `'sum'`: the output will be summed. Note: `size_average` and `reduce` are in the process of being deprecated, and in the meantime, specifying either of those two args will override `reduction`. Default: `'mean'`

Example:

```
>>> # input is of size N x C = 3 x 5
>>> input = torch.randn(3, 5, requires_grad=True)
>>> # each element in target has to have 0 <= value < C
>>> target = torch.tensor([1, 0, 4])
>>> output = F.nll_loss(F.log_softmax(input), target)
>>> output.backward()
```

17.9.16 smooth_l1_loss

torch.nn.functional.**smooth_l1_loss**(*input*, *target*, *size_average=None*, *reduce=None*, *reduction='mean'*)

Function that uses a squared term if the absolute element-wise error falls below 1 and an L1 term otherwise.

See *SmoothL1Loss* for details.

17.9.17 soft_margin_loss

torch.nn.functional.**soft_margin_loss**(*input*, *target*, *size_average=None*, *reduce=None*, *reduction='mean'*) → Tensor

See *SoftMarginLoss* for details.

17.9.18 triplet_margin_loss

torch.nn.functional.**triplet_margin_loss**(*anchor*, *positive*, *negative*, *margin=1.0*, *p=2*, *eps=1e-06*, *swap=False*, *size_average=None*, *reduce=None*, *reduction='mean'*)

See *TripletMarginLoss* for details

17.10 Vision functions

17.10.1 pixel_shuffle

torch.nn.functional.**pixel_shuffle**()

Rearranges elements in a tensor of shape $(*, C \times r^2, H, W)$ to a tensor of shape $(*, C, H \times r, W \times r)$.

See *PixelShuffle* for details.

Parameters

- **input** (Tensor) – the input tensor
- **upscale_factor** (*int*) – factor to increase spatial resolution by

Examples:

```
>>> input = torch.randn(1, 9, 4, 4)
>>> output = torch.nn.functional.pixel_shuffle(input, 3)
>>> print(output.size())
torch.Size([1, 1, 12, 12])
```

17.10.2 pad

torch.nn.functional.**pad**(*input*, *pad*, *mode='constant'*, *value=0*)

Pads tensor.

Padding size: The padding size by which to pad some dimensions of `input` are described starting from the last dimension and moving forward. $\left\lfloor \frac{\text{len(pad)}}{2} \right\rfloor$ dimensions of `input` will be padded. For example, to pad only the last dimension of the input tensor, then *pad* has the form $(\text{padding\_left}, \text{padding\_right})$; to pad the last 2 dimensions of the input tensor, then use $(\text{padding\_left}, \text{padding\_right}, \text{padding\_top}, \text{padding\_bottom})$; to pad the last 3 dimensions, use $(\text{padding\_left}, \text{padding\_right}, \text{padding\_top}, \text{padding\_bottom}\ \text{padding\_front}, \text{padding\_back})$.

Padding mode: See *torch.nn.ConstantPad2d*, *torch.nn.ReflectionPad2d*, and *torch.nn.ReplicationPad2d* for concrete examples on how each of the padding modes works. Constant padding is implemented for arbitrary dimensions. Replicate padding is implemented for padding the last 3 dimensions of 5D input tensor, or the last 2 dimensions of 4D input tensor, or the last dimension of 3D input tensor. Reflect padding is only implemented for padding the last 2 dimensions of 4D input tensor, or the last dimension of 3D input tensor.

Note: When using the CUDA backend, this operation may induce nondeterministic behaviour in its backward pass that is not easily switched off. Please see the notes on *Reproducibility* for background.

Parameters

- **input** (Tensor) – N-dimensional tensor
- **pad** (*tuple*) – m-elements tuple, where $\frac{m}{2} \leq$ input dimensions and m is even.
- **mode** – `'constant'`, `'reflect'`, `'replicate'` or `'circular'`. Default: `'constant'`
- **value** – fill value for `'constant'` padding. Default: `0`

Examples:

```
>>> t4d = torch.empty(3, 3, 4, 2)
>>> p1d = (1, 1) # pad last dim by 1 on each side
>>> out = F.pad(t4d, p1d, "constant", 0)  # effectively zero padding
>>> print(out.data.size())
torch.Size([3, 3, 4, 4])
>>> p2d = (1, 1, 2, 2) # pad last dim by (1, 1) and 2nd to last by (2, 2)
```

(continues on next page)

(continued from previous page)

```
>>> out = F.pad(t4d, p2d, "constant", 0)
>>> print(out.data.size())
torch.Size([3, 3, 8, 4])
>>> t4d = torch.empty(3, 3, 4, 2)
>>> p3d = (0, 1, 2, 1, 3, 3) # pad by (0, 1), (2, 1), and (3, 3)
>>> out = F.pad(t4d, p3d, "constant", 0)
>>> print(out.data.size())
torch.Size([3, 9, 7, 3])
```

17.10.3 interpolate

torch.nn.functional.**interpolate**(*input*, *size=None*, *scale_factor=None*, *mode='nearest'*, *align_corners=None*)

Down/up samples the input to either the given `size` or the given `scale_factor`

The algorithm used for interpolation is determined by `mode`.

Currently temporal, spatial and volumetric sampling are supported, i.e. expected inputs are 3-D, 4-D or 5-D in shape.

The input dimensions are interpreted in the form: *mini-batch x channels x [optional depth] x [optional height] x width*.

The modes available for resizing are: *nearest*, *linear* (3D-only), *bilinear*, *bicubic* (4D-only), *trilinear* (5D-only), *area*

Parameters

- **input** (Tensor) – the input tensor
- **size** (*int or Tuple[int] or Tuple[int, int] or Tuple[int, int, int]*) – output spatial size.
- **scale_factor** (*float or Tuple[float]*) – multiplier for spatial size. Has to match input size if it is a tuple.
- **mode** (*str*) – algorithm used for upsampling: `'nearest'|'linear'|'bilinear'|'bicubic'|'trilinear'|'area'`. Default: `'nearest'`
- **align_corners** (*bool, optional*) – Geometrically, we consider the pixels of the input and output as squares rather than points. If set to `True`, the input and output tensors are aligned by the center points of their corner pixels. If set to `False`, the input and output tensors are aligned by the corner points of their corner pixels, and the interpolation uses edge value padding for out-of-boundary values. This only has effect when `mode` is `'linear'`, `'bilinear'`, `'bicubic'`, or `'trilinear'`. Default: `False`

Warning: With `align_corners = True`, the linearly interpolating modes (*linear*, *bilinear*, and *trilinear*) dont proportionally align the output and input pixels, and thus the output values can depend on the input size. This was the default behavior for these modes up to version 0.3.1. Since then, the default behavior is `align_corners = False`. See *Upsample* for concrete examples on how this affects the outputs.

Note: When using the CUDA backend, this operation may induce nondeterministic behaviour in its backward pass that is not easily switched off. Please see the notes on *Reproducibility* for background.

17.10.4 upsample

torch.nn.functional.**upsample**(*input*, *size=None*, *scale_factor=None*, *mode='nearest'*, *align_corners=None*)

Upsamples the input to either the given `size` or the given `scale_factor`

Warning: This function is deprecated in favor of *torch.nn.functional.interpolate()*. This is equivalent with `nn.functional.interpolate(...)`.

Note: When using the CUDA backend, this operation may induce nondeterministic behaviour in its backward pass that is not easily switched off. Please see the notes on *Reproducibility* for background.

The algorithm used for upsampling is determined by `mode`.

Currently temporal, spatial and volumetric upsampling are supported, i.e. expected inputs are 3-D, 4-D or 5-D in shape.

The input dimensions are interpreted in the form: *mini-batch x channels x [optional depth] x [optional height] x width*.

The modes available for upsampling are: *nearest*, *linear* (3D-only), *bilinear*, *bicubic* (4D-only), *trilinear* (5D-only)

Parameters

- **input** (*Tensor*) – the input tensor
- **size** (*int or Tuple[int] or Tuple[int, int] or Tuple[int, int, int]*) – output spatial size.
- **scale_factor** (*float or Tuple[float]*) – multiplier for spatial size. Has to be an integer.
- **mode** (*string*) – algorithm used for upsampling: `'nearest' | 'linear' | 'bilinear' | 'bicubic' | 'trilinear'`. Default: `'nearest'`
- **align_corners** (*bool, optional*) – Geometrically, we consider the pixels of the input and output as squares rather than points. If set to `True`, the input and output tensors are aligned by the center points of their corner pixels. If set to `False`, the input and output tensors are aligned by the corner points of their corner pixels, and the interpolation uses edge value padding for out-of-boundary values. This only has effect when `mode` is `'linear'`, `'bilinear'`, `'bicubic'` or `'trilinear'`. Default: `False`

Warning: With `align_corners = True`, the linearly interpolating modes (*linear*, *bilinear*, and *trilinear*) dont proportionally align the output and input pixels, and thus the output values can depend on the input size. This was the default behavior for these modes up to version 0.3.1. Since then, the default behavior is `align_corners = False`. See *Upsample* for concrete examples on how this affects the outputs.

17.10.5 upsample_nearest

torch.nn.functional.**upsample_nearest**(*input*, *size=None*, *scale_factor=None*)

Upsamples the input, using nearest neighbours pixel values.

> **Warning:** This function is deprecated in favor of *torch.nn.functional.interpolate()*. This is equivalent with `nn.functional.interpolate(..., mode='nearest')`.

Currently spatial and volumetric upsampling are supported (i.e. expected inputs are 4 or 5 dimensional).

Parameters

- **input** (Tensor) – input
- **size** (*int or Tuple[int, int] or Tuple[int, int, int]*) – output spatia size.
- **scale_factor** (*int*) – multiplier for spatial size. Has to be an integer.

Note: When using the CUDA backend, this operation may induce nondeterministic behaviour in its backward pass that is not easily switched off. Please see the notes on *Reproducibility* for background.

17.10.6 upsample_bilinear

`torch.nn.functional.`**upsample_bilinear**(*input*, *size=None*, *scale_factor=None*)

Upsamples the input, using bilinear upsampling.

> **Warning:** This function is deprecated in favor of *torch.nn.functional.interpolate()*. This is equivalent with `nn.functional.interpolate(..., mode='bilinear', align_corners=True)`.

Expected inputs are spatial (4 dimensional). Use *upsample_trilinear* fo volumetric (5 dimensional) inputs.

Parameters

- **input** (Tensor) – input
- **size** (*int or Tuple[int, int]*) – output spatial size.
- **scale_factor** (*int or Tuple[int, int]*) – multiplier for spatial size

---

Note: When using the CUDA backend, this operation may induce nondeterministic behaviour in its backward pass that is not easily switched off. Please see the notes on *Reproducibility* for background.

17.10.7 grid_sample

`torch.nn.functional.`**grid_sample**(*input*, *grid*, *mode='bilinear'*, *padding_mode='zeros'*)

Given an `input` and a flow-field `grid`, computes the `output` using `input` values and pixel locations from `grid`.

Currently, only spatial (4-D) and volumetric (5-D) `input` are supported.

In the spatial (4-D) case, for `input` with shape $(N, C, H_{\text{in}}, W_{\text{in}})$ and `grid` with shape $(N, H_{\text{out}}, W_{\text{out}}, 2)$, the output will have shape $(N, C, H_{\text{out}}, W_{\text{out}})$.

For each output location `output[n, :, h, w]`, the size-2 vector `grid[n, h, w]` specifies `input` pixel locations `x` and `y`, which are used to interpolate the output value `output[n, :, h, w]`. In the

case of 5D inputs, `grid[n, d, h, w]` specifies the `x`, `y`, `z` pixel locations for interpolating `output[n, :, d, h, w]`. `mode` argument specifies `nearest` or `bilinear` interpolation method to sample the input pixels.

`grid` should have most values in the range of `[-1, 1]`. This is because the pixel locations are normalized by the `input` spatial dimensions. For example, values `x = -1, y = -1` is the left-top pixel of `input`, and values `x = 1, y = 1` is the right-bottom pixel of `input`.

If `grid` has values outside the range of `[-1, 1]`, those locations are handled as defined by `padding_mode`. Options are

- `padding_mode="zeros"`: use `0` for out-of-bound values,
- `padding_mode="border"`: use border values for out-of-bound values,
- `padding_mode="reflection"`: use values at locations reflected by the border for out-of-bound values. For location far away from the border, it will keep being reflected until becoming in bound, e.g., (normalized) pixel location `x = -3.5` reflects by `-1` and becomes `x' = 1.5`, then reflects by border `1` and becomes `x'' = -0.5`.

Note: This function is often used in building Spatial Transformer Networks.

Note: When using the CUDA backend, this operation may induce nondeterministic behaviour in its backward pass that is not easily switched off. Please see the notes on *Reproducibility* for background.

Parameters

- **input** (Tensor) – input of shape $(N, C, H_{\text{in}}, W_{\text{in}})$ (4-D case) or $(N, C, D_{\text{in}}, H_{\text{in}}, W_{\text{in}})$ (5-D case)
- **grid** (Tensor) – flow-field of shape $(N, H_{\text{out}}, W_{\text{out}}, 2)$ (4-D case) or $(N, D_{\text{out}}, H_{\text{out}}, W_{\text{out}}, 3)$ (5-D case)
- **mode** (*str*) – interpolation mode to calculate output values `'bilinear'` | `'nearest'`. Default: `'bilinear'`
- **padding_mode** (*str*) – padding mode for outside grid values `'zeros'` | `'border'` | `'reflection'`. Default: `'zeros'`

Returns output Tensor

Return type output (*Tensor*)

17.10.8 affine_grid

torch.nn.functional.**affine_grid**(*theta*, *size*)

Generates a 2d flow field, given a batch of affine matrices `theta`. Generally used in conjunction with *grid_sample()* to implement Spatial Transformer Networks.

Parameters

- **theta** (Tensor) – input batch of affine matrices ($N \times 2 \times 3$)
- **size** (*torch.Size*) – the target output image size ($N \times C \times H \times W$). Example: torch.Size((32, 3, 24, 24))

Returns output Tensor of size ($N \times H \times W \times 2$)

Return type output (*Tensor*)

17.11 DataParallel functions (multi-GPU, distributed)

17.11.1 data_parallel

torch.nn.parallel.**data_parallel**(*module*, *inputs*, *device_ids=None*, *output_device=None*, *dim=0*, *module_kwargs=None*)

Evaluates module(input) in parallel across the GPUs given in device_ids.

This is the functional version of the DataParallel module.

Parameters

- **module** (Module) – the module to evaluate in parallel
- **inputs** (Tensor) – inputs to the module
- **device_ids** (*list of python:int or* torch.device) – GPU ids on which to replicate module
- **output_device** (*list of python:int or* torch.device) – GPU location of the output Use -1 to indicate the CPU. (default: device_ids[0])

Returns a Tensor containing the result of module(input) located on output_device

CHAPTER

EIGHTEEN

TORCH.TENSOR

A *torch.Tensor* is a multi-dimensional matrix containing elements of a single data type.

Torch defines nine CPU tensor types and nine GPU tensor types:

Data type	dtype	CPU tensor	GPU tensor
32-bit floating point	`torch.float32` or `torch.float`	`torch.FloatTensor`	`torch.cuda.FloatTensor`
64-bit floating point	`torch.float64` or `torch.double`	`torch.DoubleTensor`	`torch.cuda.DoubleTensor`
16-bit floating point	`torch.float16` or `torch.half`	`torch.HalfTensor`	`torch.cuda.HalfTensor`
8-bit integer (unsigned)	`torch.uint8`	`torch.ByteTensor`	`torch.cuda.ByteTensor`
8-bit integer (signed)	`torch.int8`	`torch.CharTensor`	`torch.cuda.CharTensor`
16-bit integer (signed)	`torch.int16` or `torch.short`	`torch.ShortTensor`	`torch.cuda.ShortTensor`
32-bit integer (signed)	`torch.int32` or `torch.int`	`torch.IntTensor`	`torch.cuda.IntTensor`
64-bit integer (signed)	`torch.int64` or `torch.long`	`torch.LongTensor`	`torch.cuda.LongTensor`
Boolean	`torch.bool`	*torch.BoolTensor*	`torch.cuda.BoolTensor`

torch.Tensor is an alias for the default tensor type (`torch.FloatTensor`).

A tensor can be constructed from a Python `list` or sequence using the *torch.tensor()* constructor:

```
>>> torch.tensor([[1., -1.], [1., -1.]])
tensor([[ 1.0000, -1.0000],
        [ 1.0000, -1.0000]])
>>> torch.tensor(np.array([[1, 2, 3], [4, 5, 6]]))
tensor([[ 1,  2,  3],
        [ 4,  5,  6]])
```

Warning: *torch.tensor()* always copies `data`. If you have a Tensor `data` and just want to change its `requires_grad` flag, use *requires_grad_()* or *detach()* to avoid a copy. If you have a numpy array and want to avoid a copy, use *torch.as_tensor()*.

A tensor of specific data type can be constructed by passing a *torch.dtype* and/or a *torch.device* to a constructor or tensor creation op:

```
>>> torch.zeros([2, 4], dtype=torch.int32)
tensor([[ 0,  0,  0,  0],
        [ 0,  0,  0,  0]], dtype=torch.int32)
>>> cuda0 = torch.device('cuda:0')
>>> torch.ones([2, 4], dtype=torch.float64, device=cuda0)
tensor([[ 1.0000,  1.0000,  1.0000,  1.0000],
        [ 1.0000,  1.0000,  1.0000,  1.0000]], dtype=torch.float64, device='cuda:0')
```

The contents of a tensor can be accessed and modified using Pythons indexing and slicing notation:

```
>>> x = torch.tensor([[1, 2, 3], [4, 5, 6]])
>>> print(x[1][2])
tensor(6)
>>> x[0][1] = 8
>>> print(x)
tensor([[ 1,  8,  3],
        [ 4,  5,  6]])
```

Use *torch.Tensor.item()* to get a Python number from a tensor containing a single value:

```
>>> x = torch.tensor([[1]])
>>> x
tensor([[ 1]])
>>> x.item()
1
>>> x = torch.tensor(2.5)
>>> x
tensor(2.5000)
>>> x.item()
2.5
```

A tensor can be created with `requires_grad=True` so that *torch.autograd* records operations on them for automatic differentiation.

```
>>> x = torch.tensor([[1., -1.], [1., 1.]], requires_grad=True)
>>> out = x.pow(2).sum()
>>> out.backward()
>>> x.grad
tensor([[ 2.0000, -2.0000],
        [ 2.0000,  2.0000]])
```

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.

Note: For more information on the *torch.dtype*, *torch.device*, and *torch.layout* attributes of a *torch.Tensor*, see *Tensor Attributes*.

Note: 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.

Note: To change an existing tensors *torch.device* and/or *torch.dtype*, consider using *to()* method on the tensor.

Warning: Current implementation of *torch.Tensor* introduces memory overhead, thus it might lead to unexpectedly high memory usage in the applications with many tiny tensors. If this is your case, consider using one large structure.

class torch.**Tensor**

There are a few main ways to create a tensor, depending on your use case.

- To create a tensor with pre-existing data, use *torch.tensor()*.
- To create a tensor with specific size, use `torch.*` tensor creation ops (see *Creation Ops*).
- To create a tensor with the same size (and similar types) as another tensor, use `torch.*_like` tensor creation ops (see *Creation Ops*).
- To create a tensor with similar type but different size as another tensor, use `tensor.new_*` creation ops.

new_tensor(*data*, *dtype=None*, *device=None*, *requires_grad=False*) → Tensor

Returns a new Tensor with `data` as the tensor data. By default, the returned Tensor has the same *torch.dtype* and *torch.device* as this tensor.

Warning: *new_tensor()* always copies `data`. If you have a Tensor `data` and want to avoid a copy, use *torch.Tensor.requires_grad_()* or *torch.Tensor.detach()*. If you have a numpy array and want to avoid a copy, use *torch.from_numpy()*.

Warning: When data is a tensor *x*, *new_tensor()* reads out the data from whatever it is passed, and constructs a leaf variable. Therefore `tensor.new_tensor(x)` is equivalent to `x.clone().detach()` and `tensor.new_tensor(x, requires_grad=True)` is equivalent to `x.clone().detach().requires_grad_(True)`. The equivalents using `clone()` and `detach()` are recommended.

Parameters

- **data** (*array_like*) – The returned Tensor copies `data`.
- **dtype** (*torch.dtype*, optional) – the desired type of returned tensor. Default: if None, same *torch.dtype* as this tensor.
- **device** (*torch.device*, optional) – the desired device of returned tensor. Default: if None, same *torch.device* as this tensor.
- **requires_grad** (*bool, optional*) – If autograd should record operations on the returned tensor. Default: `False`.

Example:

```
>>> tensor = torch.ones((2,), dtype=torch.int8)
>>> data = [[0, 1], [2, 3]]
>>> tensor.new_tensor(data)
```

(continues on next page)

(continued from previous page)

```
tensor([[ 0,  1],
        [ 2,  3]], dtype=torch.int8)
```

new_full(*size*, *fill_value*, *dtype=None*, *device=None*, *requires_grad=False*) → Tensor

Returns a Tensor of size *size* filled with `fill_value`. By default, the returned Tensor has the same *torch.dtype* and *torch.device* as this tensor.

Parameters

- **fill_value** (*scalar*) – the number to fill the output tensor with.
- **dtype** (*torch.dtype*, optional) – the desired type of returned tensor. Default: if None, same *torch.dtype* as this tensor.
- **device** (*torch.device*, optional) – the desired device of returned tensor. Default: if None, same *torch.device* as this tensor.
- **requires_grad** (*bool, optional*) – If autograd should record operations on the returned tensor. Default: `False`.

Example:

```
>>> tensor = torch.ones((2,), dtype=torch.float64)
>>> tensor.new_full((3, 4), 3.141592)
tensor([[ 3.1416,  3.1416,  3.1416,  3.1416],
        [ 3.1416,  3.1416,  3.1416,  3.1416],
        [ 3.1416,  3.1416,  3.1416,  3.1416]], dtype=torch.float64)
```

new_empty(*size*, *dtype=None*, *device=None*, *requires_grad=False*) → Tensor

Returns a Tensor of size *size* filled with uninitialized data. By default, the returned Tensor has the same *torch.dtype* and *torch.device* as this tensor.

Parameters

- **dtype** (*torch.dtype*, optional) – the desired type of returned tensor. Default: if None, same *torch.dtype* as this tensor.
- **device** (*torch.device*, optional) – the desired device of returned tensor. Default: if None, same *torch.device* as this tensor.
- **requires_grad** (*bool, optional*) – If autograd should record operations on the returned tensor. Default: `False`.

Example:

```
>>> tensor = torch.ones(())
>>> tensor.new_empty((2, 3))
tensor([[ 5.8182e-18,  4.5765e-41, -1.0545e+30],
        [ 3.0949e-41,  4.4842e-44,  0.0000e+00]])
```

new_ones(*size*, *dtype=None*, *device=None*, *requires_grad=False*) → Tensor

Returns a Tensor of size *size* filled with `1`. By default, the returned Tensor has the same *torch.dtype* and *torch.device* as this tensor.

Parameters

- **size** (*int...*) – a list, tuple, or `torch.Size` of integers defining the shape of the output tensor.
- **dtype** (*torch.dtype*, optional) – the desired type of returned tensor. Default: if None, same *torch.dtype* as this tensor.

- **device** (*torch.device*, optional) – the desired device of returned tensor. Default: if None, same *torch.device* as this tensor.
- **requires_grad** (*bool, optional*) – If autograd should record operations on the returned tensor. Default: `False`.

Example:

```
>>> tensor = torch.tensor((), dtype=torch.int32)
>>> tensor.new_ones((2, 3))
tensor([[ 1,  1,  1],
        [ 1,  1,  1]], dtype=torch.int32)
```

new_zeros(*size*, *dtype=None*, *device=None*, *requires_grad=False*) → Tensor

Returns a Tensor of size *size* filled with `0`. By default, the returned Tensor has the same *torch.dtype* and *torch.device* as this tensor.

Parameters

- **size** (*int...*) – a list, tuple, or `torch.Size` of integers defining the shape of the output tensor.
- **dtype** (*torch.dtype*, optional) – the desired type of returned tensor. Default: if None, same *torch.dtype* as this tensor.
- **device** (*torch.device*, optional) – the desired device of returned tensor. Default: if None, same *torch.device* as this tensor.
- **requires_grad** (*bool, optional*) – If autograd should record operations on the returned tensor. Default: `False`.

Example:

```
>>> tensor = torch.tensor((), dtype=torch.float64)
>>> tensor.new_zeros((2, 3))
tensor([[ 0.,  0.,  0.],
        [ 0.,  0.,  0.]], dtype=torch.float64)
```

is_cuda

Is `True` if the Tensor is stored on the GPU, `False` otherwise.

device

Is the *torch.device* where this Tensor is.

grad

This attribute is `None` by default and becomes a Tensor the first time a call to *backward()* computes gradients for `self`. The attribute will then contain the gradients computed and future calls to *backward()* will accumulate (add) gradients into it.

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*) → Tensor
add(value=1, other) -> Tensor

See `torch.add()`

add_(*value*) → Tensor
add_(value=1, other) -> Tensor

In-place version of `add()`

addbmm(*beta=1*, *alpha=1*, *batch1*, *batch2*) → Tensor
See `torch.addbmm()`

addbmm_(*beta=1*, *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*, *alpha=1*, *mat1*, *mat2*) → Tensor
See `torch.addmm()`

addmm_(*beta=1*, *alpha=1*, *mat1*, *mat2*) → Tensor
In-place version of `addmm()`

addmv(*beta=1*, *alpha=1*, *mat*, *vec*) → Tensor
See `torch.addmv()`

addmv_(*beta=1*, *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()`

allclose(*other*, *rtol=1e-05*, *atol=1e-08*, *equal_nan=False*) → Tensor
See `torch.allclose()`

apply_(*callable*) → Tensor
Applies the function `callable` to each element in the tensor, replacing each element with the value returned by `callable`.

Note: This function only works with CPU tensors and should not be used in code sections that require high performance.

---

argmax(*dim=None*, *keepdim=False*) → LongTensor
See `torch.argmax()`

argmin(*dim=None*, *keepdim=False*) → LongTensor
See `torch.argmin()`

argsort(*dim=-1*, *descending=False*) → LongTensor
See :func: *torch.argsort*

asin() → Tensor
See *torch.asin()*

asin_() → Tensor
In-place version of *asin()*

as_strided()

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()*

backward(*gradient=None*, *retain_graph=None*, *create_graph=False*)
Computes the gradient of current tensor w.r.t. graph leaves.

The graph is differentiated using the chain rule. If the tensor is non-scalar (i.e. its data has more than one element) and requires gradient, the function additionally 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.

Parameters

- **gradient** (*Tensor or None*) – Gradient w.r.t. the tensor. If it is a tensor, it will be automatically converted to a Tensor that does not require grad unless `create_graph` is True. None values can be specified for scalar Tensors or ones that dont require grad. If a None value would be acceptable then this argument is optional.
- **retain_graph** (*bool, optional*) – If `False`, the graph used to compute the grads will be freed. Note that in nearly all cases setting this option to True is not needed and often can be worked around in a much more efficient way. Defaults to the value of `create_graph`.
- **create_graph** (*bool, optional*) – If `True`, graph of the derivative will be constructed, allowing to compute higher order derivative products. Defaults to `False`.

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(*, *generator=None*) → Tensor
Returns a result tensor where each `result[i]` is independently sampled from Bernoulli(`self[i]`). `self` must have floating point `dtype`, and the result will have the same `dtype`.

See *torch.bernoulli()*

bernoulli_()

bernoulli_(*p=0.5*, *, *generator=None*) → Tensor
Fills each location of `self` with an independent sample from Bernoulli(`p`). `self` can have integral `dtype`.

bernoulli_(*p_tensor*, *, *generator=None*) → Tensor
`p_tensor` should be a tensor containing probabilities to be used for drawing the binary random number.

The i^{th} element of `self` tensor will be set to a value sampled from Bernoulli(`p_tensor[i]`).

`self` can have integral `dtype`, but `p_tensor` must have floating point `dtype`.

See also *bernoulli()* and *torch.bernoulli()*

bincount(*weights=None*, *minlength=0*) → Tensor
See *torch.bincount()*

bmm(*batch2*) → Tensor
See *torch.bmm()*

byte() → Tensor
`self.byte()` is equivalent to `self.to(torch.uint8)`. See *to()*.

cauchy_(*median=0*, *sigma=1*, *, *generator=None*) → Tensor
Fills the tensor with numbers drawn from the Cauchy distribution:

$$f(x) = \frac{1}{\pi} \frac{\sigma}{(x - \text{median})^2 + \sigma^2}$$

ceil() → Tensor
See *torch.ceil()*

ceil_() → Tensor
In-place version of *ceil()*

char() → Tensor
`self.char()` is equivalent to `self.to(torch.int8)`. See *to()*.

cholesky(*upper=False*) → Tensor
See *torch.cholesky()*

cholesky_solve(*input2*, *upper=False*) → Tensor
See *torch.cholesky_solve()*

chunk(*chunks*, *dim=0*) → List 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 `self` tensor. The copy has the same size and data type as `self`.

Note: Unlike *copy_()*, this function is recorded in the computation graph. Gradients propagating to the cloned tensor will propagate to the original tensor.

---

contiguous() → Tensor
Returns a contiguous tensor containing the same data as `self` tensor. If `self` tensor is contiguous, this function returns the `self` tensor.

copy_(*src*, *non_blocking=False*) → Tensor
Copies the elements from `src` into `self` tensor and returns `self`.

The `src` tensor must be *broadcastable* with the `self` tensor. It may be of a different data type or reside on a different device.

Parameters

- **src** (`Tensor`) – the source tensor to copy from
- **non_blocking** (*bool*) – if `True` and this copy is between CPU and GPU, the copy may occur asynchronously with respect to the host. For other cases, 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() → Tensor
Returns a copy of this object in CPU memory.

If this object is already in CPU memory and on the correct device, then no copy is performed and the original object is returned.

cross(*other*, *dim=-1*) → Tensor
See *torch.cross()*

cuda(*device=None*, *non_blocking=False*) → Tensor
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.

Parameters

- **device** (*torch.device*) – The destination GPU device. Defaults to the current CUDA device.
- **non_blocking** (*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. Default: `False`.

cumprod(*dim*, *dtype=None*) → Tensor
See *torch.cumprod()*

cumsum(*dim*, *dtype=None*) → Tensor
See *torch.cumsum()*

data_ptr() → int
Returns the address of the first element of `self` tensor.

dequantize() → Tensor
Given a quantized Tensor, dequantize it and return the dequantized float Tensor.

det() → Tensor
See *torch.det()*

dense_dim() → int
If `self` is a sparse COO tensor (i.e., with `torch.sparse_coo` layout), this returns a the number of dense dimensions. Otherwise, this throws an error.

See also *Tensor.sparse_dim()*.

detach()
Returns a new Tensor, detached from the current graph.

The result will never require gradient.

Note: Returned Tensor shares the same storage with the original one. In-place modifications on either of them will be seen, and may trigger errors in correctness checks. IMPORTANT NOTE: Previously, in-place size / stride / storage changes (such as *resize_* / *resize_as_* / *set_* / *transpose_*) to the returned tensor also update the original tensor. Now, these in-place changes will not update the original tensor anymore, and will instead trigger an error. For sparse tensors: In-place indices / values changes (such as *zero_* / *copy_* / *add_*) to the returned tensor will not update the original tensor anymore, and will instead trigger an error.

---

detach_()
Detaches the Tensor from the graph that created it, making it a leaf. Views cannot be detached in-place.

diag(*diagonal=0*) → Tensor
See *torch.diag()*

diag_embed(*offset=0*, *dim1=-2*, *dim2=-1*) → Tensor
See *torch.diag_embed()*

diagflat(*diagonal=0*) → Tensor
See *torch.diagflat()*

diagonal(*offset=0*, *dim1=0*, *dim2=1*) → Tensor
See *torch.diagonal()*

digamma() → Tensor
See *torch.digamma()*

digamma_() → Tensor
In-place version of *digamma()*

dim() → int
Returns the number of dimensions of `self` tensor.

dist(*other*, *p=2*) → Tensor
See *torch.dist()*

div(*value*) → Tensor
See *torch.div()*

div_(*value*) → Tensor
In-place version of *div()*

dot(*tensor2*) → Tensor
See *torch.dot()*

double() → Tensor
`self.double()` is equivalent to `self.to(torch.float64)`. See *to()*.

eig(*eigenvectors=False) -> (Tensor*, *Tensor*)
See *torch.eig()*

element_size() → int
Returns the size in bytes of an individual element.

Example:

```
>>> torch.tensor([]).element_size()
4
>>> torch.tensor([], dtype=torch.uint8).element_size()
1
```

eq(*other*) → Tensor
See *torch.eq()*

eq_(*other*) → Tensor
In-place version of *eq()*

equal(*other*) → bool
See *torch.equal()*

erf() → Tensor
See *torch.erf()*

erf_() → Tensor
In-place version of *erf()*

erfc() → Tensor
See *torch.erfc()*

erfc_() → Tensor
In-place version of *erfc()*

erfinv() → Tensor
See *torch.erfinv()*

erfinv_() → Tensor
In-place version of *erfinv()*

exp() → Tensor
See *torch.exp()*

exp_() → Tensor
In-place version of *exp()*

expm1() → Tensor
See *torch.expm1()*

expm1_() → Tensor
In-place version of *expm1()*

expand(**sizes*) → Tensor
Returns a new view of the `self` tensor with singleton dimensions expanded to a larger size.

Passing -1 as the size for a dimension means not changing the size of that dimension.

Tensor can be also expanded to a larger number of dimensions, and the new ones will be appended at the front. For the new dimensions, the size cannot be set to -1.

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.

Parameters ***sizes** (*torch.Size or int...*) – the desired expanded size

> **Warning:** More than one element of an expanded tensor may refer to a single memory location. As a result, in-place operations (especially ones that are vectorized) may result in incorrect behavior. If you need to write to the tensors, please clone them first.

Example:

```
>>> x = torch.tensor([[1], [2], [3]])
>>> x.size()
torch.Size([3, 1])
>>> x.expand(3, 4)
tensor([[ 1,  1,  1,  1],
        [ 2,  2,  2,  2],
        [ 3,  3,  3,  3]])
>>> x.expand(-1, 4)   # -1 means not changing the size of that dimension
tensor([[ 1,  1,  1,  1],
        [ 2,  2,  2,  2],
        [ 3,  3,  3,  3]])
```

expand_as(*other*) → Tensor
Expand this tensor to the same size as `other`. `self.expand_as(other)` is equivalent to `self.expand(other.size())`.

Please see *expand()* for more information about `expand`.

Parameters **other** (*torch.Tensor*) – The result tensor has the same size as `other`.

exponential_(*lambd=1*, *, *generator=None*) → Tensor
Fills `self` tensor with elements drawn from the exponential distribution:

$$f(x) = \lambda e^{-\lambda x}$$

fft(*signal_ndim*, *normalized=False*) → Tensor
See *torch.fft()*

fill_(*value*) → Tensor
Fills `self` tensor with the specified value.

flatten(*input*, *start_dim=0*, *end_dim=-1*) → Tensor
see *torch.flatten()*

flip(*dims*) → Tensor
See *torch.flip()*

float() → Tensor
`self.float()` is equivalent to `self.to(torch.float32)`. See *to()*.

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()*

geometric_(*p*, *, *generator=None*) → Tensor
Fills `self` tensor with elements drawn from the geometric distribution:

$$f(X = k) = (1 - p)^{k-1} p$$

geqrf() -> (*Tensor*, *Tensor*)
See *torch.geqrf()*

ger(*vec2*) → Tensor
See *torch.ger()*

get_device() -> *Device ordinal (Integer)*
For CUDA tensors, this function returns the device ordinal of the GPU on which the tensor resides. For CPU tensors, an error is thrown.

Example:

```
>>> x = torch.randn(3, 4, 5, device='cuda:0')
>>> x.get_device()
0
>>> x.cpu().get_device()  # RuntimeError: get_device is not implemented for␣
↪type torch.FloatTensor
```

gt(*other*) → Tensor
See *torch.gt()*

gt_(*other*) → Tensor
In-place version of *gt()*

half() → Tensor
`self.half()` is equivalent to `self.to(torch.float16)`. See *to()*.

hardshrink(*lambd=0.5*) → Tensor
See *torch.nn.functional.hardshrink()*

histc(*bins=100*, *min=0*, *max=0*) → Tensor
See *torch.histc()*

ifft(*signal_ndim*, *normalized=False*) → Tensor
See *torch.ifft()*

index_add_(*dim*, *index*, *tensor*) → Tensor
Accumulate the elements of *tensor* into the `self` tensor by adding to the indices in the order given in `index`. For example, if `dim == 0` and `index[i] == j`, then the `i`th row of *tensor* is added to the `j`th row of `self`.

The *dim*th dimension of *tensor* must have the same size as the length of `index` (which must be a vector), and all other dimensions must match `self`, or an error will be raised.

Note: When using the CUDA backend, this operation may induce nondeterministic behaviour that is not easily switched off. Please see the notes on *Reproducibility* for background.

Parameters

- **dim** (*int*) – dimension along which to index
- **index** (*LongTensor*) – indices of *tensor* to select from
- **tensor** (Tensor) – the tensor containing values to add

Example:

```
>>> x = torch.ones(5, 3)
>>> t = torch.tensor([[1, 2, 3], [4, 5, 6], [7, 8, 9]], dtype=torch.float)
>>> index = torch.tensor([0, 4, 2])
>>> x.index_add_(0, index, t)
tensor([[  2.,   3.,   4.],
        [  1.,   1.,   1.],
        [  8.,   9.,  10.],
        [  1.,   1.,   1.],
        [  5.,   6.,   7.]])
```

index_add(*dim*, *index*, *tensor*) → Tensor
Out-of-place version of *torch.Tensor.index_add_()*

index_copy_(*dim*, *index*, *tensor*) → Tensor
Copies the elements of *tensor* into the `self` tensor by selecting the indices in the order given in `index`. For example, if `dim == 0` and `index[i] == j`, then the `i`th row of *tensor* is copied to the `j`th row of `self`.

The *dim*th dimension of *tensor* must have the same size as the length of `index` (which must be a vector), and all other dimensions must match `self`, or an error will be raised.

Parameters

- **dim** (*int*) – dimension along which to index
- **index** (*LongTensor*) – indices of *tensor* to select from
- **tensor** (Tensor) – the tensor containing values to copy

Example:

```
>>> x = torch.zeros(5, 3)
>>> t = torch.tensor([[1, 2, 3], [4, 5, 6], [7, 8, 9]], dtype=torch.float)
>>> index = torch.tensor([0, 4, 2])
>>> x.index_copy_(0, index, t)
tensor([[ 1.,  2.,  3.],
        [ 0.,  0.,  0.],
        [ 7.,  8.,  9.],
        [ 0.,  0.,  0.],
        [ 4.,  5.,  6.]])
```

index_copy(*dim*, *index*, *tensor*) → Tensor
Out-of-place version of *torch.Tensor.index_copy_()*

index_fill_(*dim*, *index*, *val*) → Tensor
Fills the elements of the `self` tensor with value `val` by selecting the indices in the order given in `index`.

Parameters

- **dim** (*int*) – dimension along which to index
- **index** (*LongTensor*) – indices of `self` tensor to fill in
- **val** (*float*) – the value to fill with

Example::

```
>>> x = torch.tensor([[1, 2, 3], [4, 5, 6], [7, 8, 9]], dtype=torch.float)
>>> index = torch.tensor([0, 2])
>>> x.index_fill_(1, index, -1)
tensor([[-1.,  2., -1.],
        [-1.,  5., -1.],
        [-1.,  8., -1.]])
```

index_fill(*dim*, *index*, *value*) → Tensor
Out-of-place version of *torch.Tensor.index_fill_()*

index_put_(*indices*, *value*, *accumulate=False*) → Tensor
Puts values from the tensor `value` into the tensor `self` using the indices specified in *indices* (which is a tuple of Tensors). The expression `tensor.index_put_(indices, value)` is equivalent to `tensor[indices] = value`. Returns `self`.

If `accumulate` is `True`, the elements in *tensor* are added to `self`. If accumulate is `False`, the behavior is undefined if indices contain duplicate elements.

Parameters

- **indices** (*tuple of LongTensor*) – tensors used to index into *self*.
- **value** (Tensor) – tensor of same dtype as *self*.
- **accumulate** (*bool*) – whether to accumulate into self

index_put(*indices*, *value*, *accumulate=False*) → Tensor
Out-place version of *index_put_()*

index_select(*dim*, *index*) → Tensor
See *torch.index_select()*

indices() → Tensor
If `self` is a sparse COO tensor (i.e., with `torch.sparse_coo` layout), this returns a view of the contained indices tensor. Otherwise, this throws an error.

See also *Tensor.values()*.

Note: This method can only be called on a coalesced sparse tensor. See `Tensor.coalesce()` for details.

---

int() → Tensor
`self.int()` is equivalent to `self.to(torch.int32)`. See *to()*.

inverse() → Tensor
See *torch.inverse()*

irfft(*signal_ndim*, *normalized=False*, *onesided=True*, *signal_sizes=None*) → Tensor
See *torch.irfft()*

is_contiguous() → bool
Returns True if `self` tensor is contiguous in memory in C order.

is_floating_point() → bool
Returns True if the data type of `self` is a floating point data type.

is_leaf
All Tensors that have *requires_grad* which is `False` will be leaf Tensors by convention.

For Tensors that have *requires_grad* which is `True`, they will be leaf Tensors if they were created by the user. This means that they are not the result of an operation and so `grad_fn` is None.

Only leaf Tensors will have their *grad* populated during a call to *backward()*. To get *grad* populated for non-leaf Tensors, you can use *retain_grad()*.

Example:

```
>>> a = torch.rand(10, requires_grad=True)
>>> a.is_leaf
True
>>> b = torch.rand(10, requires_grad=True).cuda()
>>> b.is_leaf
False
# b was created by the operation that cast a cpu Tensor into a cuda Tensor
>>> c = torch.rand(10, requires_grad=True) + 2
>>> c.is_leaf
False
# c was created by the addition operation
>>> d = torch.rand(10).cuda()
>>> d.is_leaf
True
# d does not require gradients and so has no operation creating it (that is␣
↪tracked by the autograd engine)
>>> e = torch.rand(10).cuda().requires_grad_()
>>> e.is_leaf
True
# e requires gradients and has no operations creating it
>>> f = torch.rand(10, requires_grad=True, device="cuda")
>>> f.is_leaf
True
# f requires grad, has no operation creating it
```

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_shared()
Checks if tensor is in shared memory.

This is always `True` for CUDA tensors.

is_signed() → bool
Returns True if the data type of `self` is a signed data type.

is_sparse

item() → number
Returns the value of this tensor as a standard Python number. This only works for tensors with one element. For other cases, see *tolist()*.

This operation is not differentiable.

Example:

```
>>> x = torch.tensor([1.0])
>>> x.item()
1.0
```

kthvalue(*k*, *dim=None*, *keepdim=False*) -> (*Tensor*, *LongTensor*)
See *torch.kthvalue()*

le(*other*) → Tensor
See *torch.le()*

le_(*other*) → Tensor
In-place version of *le()*

lerp(*end*, *weight*) → Tensor
See *torch.lerp()*

lerp_(*end*, *weight*) → Tensor
In-place version of *lerp()*

lgamma()

lgamma_()

log() → Tensor
See *torch.log()*

log_() → Tensor
In-place version of *log()*

logdet() → Tensor
See *torch.logdet()*

log10() → Tensor
See *torch.log10()*

log10_() → Tensor
In-place version of *log10()*

log1p() → Tensor
See *torch.log1p()*

log1p_() → Tensor
In-place version of *log1p()*

log2() → Tensor
See *torch.log2()*

log2_() → Tensor
In-place version of *log2()*

log_normal_(*mean=1*, *std=2*, ***, *generator=None*)
Fills `self` tensor with numbers samples from the log-normal distribution parameterized by the given mean μ and standard deviation σ. Note that *mean* and *std* are the mean and standard deviation of the underlying normal distribution, and not of the returned distribution:

$$f(x) = \frac{1}{x\sigma\sqrt{2\pi}}\, e^{-\frac{(\ln x - \mu)^2}{2\sigma^2}}$$

logsumexp(*dim*, *keepdim=False*) → Tensor
See *torch.logsumexp()*

long() → Tensor
`self.long()` is equivalent to `self.to(torch.int64)`. See *to()*.

lt(*other*) → Tensor
See *torch.lt()*

lt_(*other*) → Tensor
In-place version of *lt()*

lu(*pivot=True*, *get_infos=False*)
See *torch.lu()*

map_(*tensor*, *callable*)
Applies `callable` for each element in `self` tensor and the given *tensor* and stores the results in `self` tensor. `self` tensor and the given *tensor* must be *broadcastable*.

The `callable` should have the signature:

```
def callable(a, b) -> number
```

masked_scatter_(*mask*, *source*)
Copies elements from `source` into `self` tensor at positions where the `mask` is one. The shape of `mask` must be *broadcastable* with the shape of the underlying tensor. The `source` should have at least as many elements as the number of ones in `mask`

Parameters

- **mask** (*ByteTensor*) – the binary mask
- **source** (Tensor) – the tensor to copy from

Note: The `mask` operates on the `self` tensor, not on the given `source` tensor.

masked_scatter(*mask*, *tensor*) → Tensor
Out-of-place version of *torch.Tensor.masked_scatter_()*

masked_fill_(*mask*, *value*)
Fills elements of `self` tensor with `value` where `mask` is one. The shape of `mask` must be *broadcastable* with the shape of the underlying tensor.

Parameters

- **mask** (*ByteTensor*) – the binary mask
- **value** (*float*) – the value to fill in with

masked_fill(*mask*, *value*) → Tensor
Out-of-place version of *torch.Tensor.masked_fill_()*

masked_select(*mask*) → Tensor
See *torch.masked_select()*

matmul(*tensor2*) → Tensor
See *torch.matmul()*

matrix_power(*n*) → Tensor
See *torch.matrix_power()*

max(*dim=None*, *keepdim=False*) -> *Tensor or (Tensor*, *Tensor*)
See *torch.max()*

mean(*dim=None*, *keepdim=False*) -> *Tensor or (Tensor*, *Tensor*)
See *torch.mean()*

median(*dim=None*, *keepdim=False*) -> *(Tensor*, *LongTensor*)
See *torch.median()*

min(*dim=None*, *keepdim=False*) -> *Tensor or (Tensor*, *Tensor*)
See *torch.min()*

mm(*mat2*) → Tensor
See *torch.mm()*

mode(*dim=None*, *keepdim=False*) -> *(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*) → Tensor
See *torch.multinomial()*

mv(*vec*) → Tensor
See *torch.mv()*

mvlgamma(*p*) → Tensor
See *torch.mvlgamma()*

mvlgamma_(*p*) → Tensor
In-place version of *mvlgamma()*

narrow(*dimension*, *start*, *length*) → Tensor
See *torch.narrow()*

Example:

```
>>> x = torch.tensor([[1, 2, 3], [4, 5, 6], [7, 8, 9]])
>>> x.narrow(0, 0, 2)
tensor([[ 1,  2,  3],
        [ 4,  5,  6]])
>>> x.narrow(1, 1, 2)
tensor([[ 2,  3],
        [ 5,  6],
        [ 8,  9]])
```

narrow_copy(*dimension*, *start*, *length*) → Tensor
Same as *Tensor.narrow()* except returning a copy rather than shared storage. This is primarily for sparse tensors, which do not have a shared-storage narrow method. Calling `narrow_copy with `dimemsion > self.sparse_dim()` will return a copy with the relevant dense dimension narrowed, and `self.shape` updated accordingly.

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()*

nonzero() → LongTensor
See *torch.nonzero()*

norm(*p='fro'*, *dim=None*, *keepdim=False*, *dtype=None*)
See *torch.norm()*

normal_(*mean=0*, *std=1*, *, *generator=None*) → Tensor
Fills `self` tensor with elements samples from the normal distribution parameterized by *mean* and *std*.

numel() → int
See *torch.numel()*

numpy() → numpy.ndarray
Returns `self` tensor as a NumPy `ndarray`. This tensor and the returned `ndarray` share the same underlying storage. Changes to `self` 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*) → Tensor
Permute the dimensions of this tensor.

Parameters ***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() → Tensor
Copies the tensor to pinned memory, if its not already pinned.

pinverse() → Tensor
See *torch.pinverse()*

polygamma()

polygamma_()

pow(*exponent*) → Tensor
See *torch.pow()*

pow_(*exponent*) → Tensor
In-place version of *pow()*

prod(*dim=None*, *keepdim=False*, *dtype=None*) → Tensor
See *torch.prod()*

put_(*indices*, *tensor*, *accumulate=False*) → Tensor
Copies the elements from *tensor* into the positions specified by indices. For the purpose of indexing, the `self` tensor is treated as if it were a 1-D tensor.

If `accumulate` is `True`, the elements in *tensor* are added to `self`. If accumulate is `False`, the behavior is undefined if indices contain duplicate elements.

Parameters

- **indices** (*LongTensor*) – the indices into self
- **tensor** (`Tensor`) – the tensor containing values to copy from
- **accumulate** (*bool*) – whether to accumulate into self

Example:

```
>>> src = torch.tensor([[4, 3, 5],
                        [6, 7, 8]])
>>> src.put_(torch.tensor([1, 3]), torch.tensor([9, 10]))
tensor([[  4,   9,   5],
        [ 10,   7,   8]])
```

qr() -> (*Tensor*, *Tensor*)
See *torch.qr()*

q_scale() → float
Given a Tensor quantized by linear(affine) quantization, returns the scale of the underlying quantizer().

q_zero_point() → int
Given a Tensor quantized by linear(affine) quantization, returns the zero_point of the underlying quantizer().

random_(*from=0*, *to=None*, *, *generator=None*) → Tensor
Fills `self` tensor with numbers sampled from the discrete uniform distribution over `[from, to - 1]`. If not specified, the values are usually only bounded by `self` tensors data type. However, for floating point types, if unspecified, range will be `[0, 2^mantissa]` to ensure that every value is representable. For example, *torch.tensor(1, dtype=torch.double).random_()* will be uniform in `[0, 2^53]`.

reciprocal() → Tensor
See *torch.reciprocal()*

reciprocal_() → Tensor
In-place version of *reciprocal()*

record_stream()

register_hook(*hook*)
Registers a backward hook.

The hook will be called every time a gradient with respect to the Tensor is computed. The hook should have the following signature:

```
hook(grad) -> Tensor 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 = torch.tensor([0., 0., 0.], requires_grad=True)
>>> h = v.register_hook(lambda grad: grad * 2)  # double the gradient
>>> v.backward(torch.tensor([1., 2., 3.]))
>>> v.grad

 2
 4
 6
[torch.FloatTensor of size (3,)]

>>> h.remove()  # removes the hook
```

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*) → Tensor
Repeats this tensor along the specified dimensions.

Unlike *expand()*, this function copies the tensors data.

Warning: `torch.repeat()` behaves differently from numpy.repeat, but is more similar to numpy.tile.

Parameters **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)
tensor([[ 1,  2,  3,  1,  2,  3],
        [ 1,  2,  3,  1,  2,  3],
        [ 1,  2,  3,  1,  2,  3],
        [ 1,  2,  3,  1,  2,  3]])
>>> x.repeat(4, 2, 1).size()
torch.Size([4, 2, 3])
```

requires_grad
Is `True` if gradients need to be computed for this Tensor, `False` otherwise.

Note: The fact that gradients need to be computed for a Tensor do not mean that the *grad* attribute will be populated, see *is_leaf* for more details.

requires_grad_(*requires_grad=True*) → Tensor

Change if autograd should record operations on this tensor: sets this tensors *requires_grad* attribute in-place. Returns this tensor.

`require_grad_()`s main use case is to tell autograd to begin recording operations on a Tensor `tensor`. If `tensor` has `requires_grad=False` (because it was obtained through a DataLoader, or required preprocessing or initialization), `tensor.requires_grad_()` makes it so that autograd will begin to record operations on `tensor`.

Parameters **requires_grad** (*bool*) – If autograd should record operations on this tensor. Default: `True`.

Example:

```
>>> # Let's say we want to preprocess some saved weights and use
>>> # the result as new weights.
>>> saved_weights = [0.1, 0.2, 0.3, 0.25]
>>> loaded_weights = torch.tensor(saved_weights)
>>> weights = preprocess(loaded_weights)  # some function
>>> weights
tensor([-0.5503,  0.4926, -2.1158, -0.8303])

>>> # Now, start to record operations done to weights
>>> weights.requires_grad_()
>>> out = weights.pow(2).sum()
>>> out.backward()
>>> weights.grad
tensor([-1.1007,  0.9853, -4.2316, -1.6606])
```

reshape(**shape*) → Tensor

Returns a tensor with the same data and number of elements as `self` but with the specified shape. This method returns a view if `shape` is compatible with the current shape. See *torch.Tensor.view()* on when it is possible to return a view.

See *torch.reshape()*

Parameters **shape** (*tuple of python:ints or int...*) – the desired shape

reshape_as(*other*) → Tensor

Returns this tensor as the same shape as `other`. `self.reshape_as(other)` is equivalent to `self.reshape(other.sizes())`. This method returns a view if `other.sizes()` is compatible with the current shape. See *torch.Tensor.view()* on when it is possible to return a view.

Please see *reshape()* for more information about `reshape`.

Parameters **other** (*torch.Tensor*) – The result tensor has the same shape as `other`.

resize_(**sizes*) → Tensor

Resizes `self` 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.

Warning: This is a low-level method. The storage is reinterpreted as C-contiguous, ignoring the current strides (unless the target size equals the current size, in which case the tensor is left unchanged). For most purposes, you will instead want to use *view()*, which checks for contiguity, or *reshape()*, which copies data if needed. To change the size in-place with custom strides, see *set_()*.

Parameters **sizes** (*torch.Size or int...*) – the desired size

Example:

```
>>> x = torch.tensor([[1, 2], [3, 4], [5, 6]])
>>> x.resize_(2, 2)
tensor([[ 1,  2],
        [ 3,  4]])
```

resize_as_(*tensor*) → Tensor
Resizes the `self` tensor to be the same size as the specified *tensor*. This is equivalent to `self.resize_(tensor.size())`.

retain_grad()
Enables .grad attribute for non-leaf Tensors.

rfft(*signal_ndim*, *normalized=False*, *onesided=True*) → Tensor
See *torch.rfft()*

roll(*shifts*, *dims*) → Tensor
See *torch.roll()*

rot90(*k*, *dims*) → Tensor
See *torch.rot90()*

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(*dim*, *index*, *source*) → Tensor
Out-of-place version of *torch.Tensor.scatter_()*

scatter_(*dim*, *index*, *src*) → Tensor
Writes all values from the tensor `src` into `self` at the indices specified in the `index` tensor. For each value in `src`, its output index is specified by its index in `src` for `dimension != dim` and by the corresponding value in `index` for `dimension = dim`.

For a 3-D tensor, `self` is updated as:

```
self[index[i][j][k]][j][k] = src[i][j][k]  # if dim == 0
self[i][index[i][j][k]][k] = src[i][j][k]  # if dim == 1
self[i][j][index[i][j][k]] = src[i][j][k]  # if dim == 2
```

This is the reverse operation of the manner described in *gather()*.

`self`, `index` and `src` (if it is a Tensor) should have same number of dimensions. It is also required that `index.size(d) <= src.size(d)` for all dimensions `d`, and that `index.size(d) <= self.size(d)` for all dimensions `d != dim`.

Moreover, 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 *dim* must be unique.

Parameters

- **dim** (*int*) – the axis along which to index
- **index** (*LongTensor*) – the indices of elements to scatter, can be either empty or the same size of src. When empty, the operation returns identity
- **src** (Tensor) – the source element(s) to scatter, incase *value* is not specified
- **value** (*float*) – the source element(s) to scatter, incase *src* is not specified

Example:

```
>>> x = torch.rand(2, 5)
>>> x
tensor([[ 0.3992,  0.2908,  0.9044,  0.4850,  0.6004],
        [ 0.5735,  0.9006,  0.6797,  0.4152,  0.1732]])
>>> torch.zeros(3, 5).scatter_(0, torch.tensor([[0, 1, 2, 0, 0], [2, 0, 0, 1,␣
↪2]]), x)
tensor([[ 0.3992,  0.9006,  0.6797,  0.4850,  0.6004],
        [ 0.0000,  0.2908,  0.0000,  0.4152,  0.0000],
        [ 0.5735,  0.0000,  0.9044,  0.0000,  0.1732]])

>>> z = torch.zeros(2, 4).scatter_(1, torch.tensor([[2], [3]]), 1.23)
>>> z
tensor([[ 0.0000,  0.0000,  1.2300,  0.0000],
        [ 0.0000,  0.0000,  0.0000,  1.2300]])
```

scatter_add_ (*dim*, *index*, *other*) → Tensor

Adds all values from the tensor `other` into `self` at the indices specified in the `index` tensor in a similar fashion as *scatter_()*. For each value in `other`, it is added to an index in `self` which is specified by its index in `other` for `dimension != dim` and by the corresponding value in `index` for `dimension = dim`.

For a 3-D tensor, `self` is updated as:

```
self[index[i][j][k]][j][k] += other[i][j][k]  # if dim == 0
self[i][index[i][j][k]][k] += other[i][j][k]  # if dim == 1
self[i][j][index[i][j][k]] += other[i][j][k]  # if dim == 2
```

`self`, `index` and `other` should have same number of dimensions. It is also required that `index.size(d) <= other.size(d)` for all dimensions `d`, and that `index.size(d) <= self.size(d)` for all dimensions `d != dim`.

Moreover, 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 *dim* must be unique.

Note: When using the CUDA backend, this operation may induce nondeterministic behaviour that is not easily switched off. Please see the notes on *Reproducibility* for background.

Parameters

- **dim** (*int*) – the axis along which to index
- **index** (*LongTensor*) – the indices of elements to scatter and add, can be either empty or the same size of src. When empty, the operation returns identity.
- **other** (Tensor) – the source elements to scatter and add

Example:

```
>>> x = torch.rand(2, 5)
>>> x
tensor([[0.7404, 0.0427, 0.6480, 0.3806, 0.8328],
        [0.7953, 0.2009, 0.9154, 0.6782, 0.9620]])
>>> torch.ones(3, 5).scatter_add_(0, torch.tensor([[0, 1, 2, 0, 0], [2, 0, 0,
↪1, 2]]), x)
tensor([[1.7404, 1.2009, 1.9154, 1.3806, 1.8328],
        [1.0000, 1.0427, 1.0000, 1.6782, 1.0000],
        [1.7953, 1.0000, 1.6480, 1.0000, 1.9620]])
```

scatter_add(*dim*, *index*, *source*) → Tensor
Out-of-place version of *torch.Tensor.scatter_add_()*

select(*dim*, *index*) → Tensor
Slices the `self` tensor along the selected dimension at the given index. This function returns a tensor with the given dimension removed.

Parameters

- **dim** (*int*) – the dimension to slice
- **index** (*int*) – the index to select with

Note: *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*) → Tensor
Sets the underlying storage, size, and strides. If `source` is a tensor, `self` tensor will share the same storage and have the same size and strides as `source`. 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.

Parameters

- **source** (*Tensor or Storage*) – the tensor or storage to use
- **storage_offset** (*int, optional*) – the offset in the storage
- **size** (*torch.Size, optional*) – the desired size. Defaults to the size of the source.
- **stride** (*tuple, optional*) – 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() → Tensor
`self.short()` is equivalent to `self.to(torch.int16)`. See *to()*.

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 `self` tensor. The returned value is a subclass of `tuple`.

Example:

```
>>> torch.empty(3, 4, 5).size()
torch.Size([3, 4, 5])
```

slogdet() -> (*Tensor*, *Tensor*)
See *torch.slogdet()*

solve(*A*) → Tensor, Tensor
See *torch.solve()*

sort(*dim=-1*, *descending=False*) -> (*Tensor*, *LongTensor*)
See *torch.sort()*

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

sparse_mask(*input*, *mask*) → Tensor
Returns a new SparseTensor with values from Tensor `input` filtered by indices of `mask` and values are ignored. `input` and `mask` must have the same shape.

Parameters

- **input** (`Tensor`) – an input Tensor
- **mask** (*SparseTensor*) – a SparseTensor which we filter `input` based on its indices

Example:

```
>>> nnz = 5
>>> dims = [5, 5, 2, 2]
>>> I = torch.cat([torch.randint(0, dims[0], size=(nnz,)),
                   torch.randint(0, dims[1], size=(nnz,))], 0).reshape(2, nnz)
>>> V = torch.randn(nnz, dims[2], dims[3])
>>> size = torch.Size(dims)
>>> S = torch.sparse_coo_tensor(I, V, size).coalesce()
>>> D = torch.randn(dims)
>>> D.sparse_mask(S)
tensor(indices=tensor([[0, 0, 0, 2],
                       [0, 1, 4, 3]]),
       values=tensor([[[ 1.6550,  0.2397],
                       [-0.1611, -0.0779]],

                      [[ 0.2326, -1.0558],
```

(continues on next page)

(continued from previous page)

```
                       [ 1.4711,  1.9678]],

                      [[-0.5138, -0.0411],
                       [ 1.9417,  0.5158]],

                      [[ 0.0793,  0.0036],
                       [-0.2569, -0.1055]]]),
      size=(5, 5, 2, 2), nnz=4, layout=torch.sparse_coo)
```

sparse_dim() → int
If `self` is a sparse COO tensor (i.e., with `torch.sparse_coo` layout), this returns a the number of sparse dimensions. Otherwise, this throws an error.

See also *Tensor.dense_dim()*.

sqrt() → Tensor
See *torch.sqrt()*

sqrt_() → Tensor
In-place version of *sqrt()*

squeeze(*dim=None*) → Tensor
See *torch.squeeze()*

squeeze_(*dim=None*) → Tensor
In-place version of *squeeze()*

std(*dim=None*, *unbiased=True*, *keepdim=False*) → Tensor
See *torch.std()*

stft(*n_fft*, *hop_length=None*, *win_length=None*, *window=None*, *center=True*, *pad_mode='reflect'*, *normalized=False*, *onesided=True*)
See *torch.stft()*

> **Warning:** This function changed signature at version 0.4.1. Calling with the previous signature may cause error or return incorrect result.

storage() → torch.Storage
Returns the underlying storage.

storage_offset() → int
Returns `self` tensors 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
```

storage_type() → type
Returns the type of the underlying storage.

stride(*dim*) → tuple or int
Returns the stride of `self` tensor.

Stride is the jump necessary to go from one element to the next one in the specified dimension *dim*. A tuple of all strides is returned when no argument is passed in. Otherwise, an integer value is returned as the stride in the particular dimension *dim*.

Parameters **dim** (*int, optional*) – the desired dimension in which stride is required

Example:

```
>>> x = torch.tensor([[1, 2, 3, 4, 5], [6, 7, 8, 9, 10]])
>>> x.stride()
(5, 1)
>>>x.stride(0)
5
>>> x.stride(-1)
1
```

sub (*value*, *other*) → Tensor
Subtracts a scalar or tensor from `self` tensor. If both `value` and `other` are specified, each element of `other` is scaled by `value` before being used.

When `other` is a tensor, the shape of `other` must be *broadcastable* with the shape of the underlying tensor.

sub_ (*x*) → Tensor
In-place version of *sub()*

sum (*dim=None*, *keepdim=False*, *dtype=None*) → Tensor
See *torch.sum()*

sum_to_size (**size*) → Tensor
Sum `this` tensor to *size*. *size* must be broadcastable to `this` tensor size. :param other: The result tensor has the same size

as `other`.

svd (*some=True*, *compute_uv=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()*

to (**args*, ***kwargs*) → Tensor
Performs Tensor dtype and/or device conversion. A *torch.dtype* and *torch.device* are inferred from the arguments of `self.to(*args, **kwargs)`.

Note: If the `self` Tensor already has the correct *torch.dtype* and *torch.device*, then `self` is returned. Otherwise, the returned tensor is a copy of `self` with the desired *torch.dtype* and *torch.device*.

Here are the ways to call `to`:

to(*dtype*, *non_blocking=False*, *copy=False*) → Tensor
Returns a Tensor with the specified `dtype`

to(*device=None*, *dtype=None*, *non_blocking=False*, *copy=False*) → Tensor
Returns a Tensor with the specified *device* and (optional) `dtype`. If `dtype` is `None` it is inferred to be `self.dtype`. When `non_blocking`, tries to convert asynchronously with respect to the host if possible, e.g., converting a CPU Tensor with pinned memory to a CUDA Tensor. When `copy` is set, a new Tensor is created even when the Tensor already matches the desired conversion.

to(*other*, *non_blocking=False*, *copy=False*) → Tensor
Returns a Tensor with same *torch.dtype* and *torch.device* as the Tensor `other`. When `non_blocking`, tries to convert asynchronously with respect to the host if possible, e.g., converting a CPU Tensor with pinned memory to a CUDA Tensor. When `copy` is set, a new Tensor is created even when the Tensor already matches the desired conversion.

Example:

```
>>> tensor = torch.randn(2, 2)  # Initially dtype=float32, device=cpu
>>> tensor.to(torch.float64)
tensor([[-0.5044,  0.0005],
        [ 0.3310, -0.0584]], dtype=torch.float64)

>>> cuda0 = torch.device('cuda:0')
>>> tensor.to(cuda0)
tensor([[-0.5044,  0.0005],
        [ 0.3310, -0.0584]], device='cuda:0')

>>> tensor.to(cuda0, dtype=torch.float64)
tensor([[-0.5044,  0.0005],
        [ 0.3310, -0.0584]], dtype=torch.float64, device='cuda:0')

>>> other = torch.randn((), dtype=torch.float64, device=cuda0)
>>> tensor.to(other, non_blocking=True)
tensor([[-0.5044,  0.0005],
        [ 0.3310, -0.0584]], dtype=torch.float64, device='cuda:0')
```

take(*indices*) → Tensor
See *torch.take()*

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()
tolist() -> list or number

Returns the tensor as a (nested) list. For scalars, a standard Python number is returned, just like with *item()*. Tensors are automatically moved to the CPU first if necessary.

This operation is not differentiable.

Examples:

```
>>> a = torch.randn(2, 2)
>>> a.tolist()
[[0.012766935862600803, 0.5415473580360413],
 [-0.08909505605697632, 0.7729271650314331]]
>>> a[0,0].tolist()
0.012766935862600803
```

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

to_sparse(*sparseDims*) → Tensor
Returns a sparse copy of the tensor. PyTorch supports sparse tensors in *coordinate format*.

Parameters **sparseDims** (*int, optional*) – the number of sparse dimensions to include in the new sparse tensor

Example:

```
>>> d = torch.tensor([[0, 0, 0], [9, 0, 10], [0, 0, 0]])
>>> d
tensor([[ 0,  0,  0],
        [ 9,  0, 10],
        [ 0,  0,  0]])
>>> d.to_sparse()
tensor(indices=tensor([[1, 1],
                       [0, 2]]),
       values=tensor([ 9, 10]),
       size=(3, 3), nnz=2, layout=torch.sparse_coo)
>>> d.to_sparse(1)
tensor(indices=tensor([[1]]),
       values=tensor([[ 9,  0, 10]]),
       size=(3, 3), nnz=1, layout=torch.sparse_coo)
```

trace() → Tensor
See *torch.trace()*

transpose(*dim0*, *dim1*) → Tensor
See *torch.transpose()*

transpose_(*dim0*, *dim1*) → Tensor
In-place version of *transpose()*

triangular_solve(*A*, *upper=True*, *transpose=False*, *unitriangular=False*) -> (*Tensor*, *Tensor*)
See *torch.triangular_solve()*

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()*

trunc() → Tensor
See *torch.trunc()*

trunc_() → Tensor
In-place version of *trunc()*

type(*dtype=None*, *non_blocking=False*, ***kwargs*) → str or Tensor
Returns the type if *dtype* is not provided, else 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.

Parameters

- **dtype** (*type or string*) – The desired type
- **non_blocking** (*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.
- ****kwargs** – For compatibility, may contain the key `async` in place of the `non_blocking` argument. The `async` arg is deprecated.

type_as(*tensor*) → 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())`

Parameters **tensor** (`Tensor`) – the tensor which has the desired type

unbind(*dim=0*) → seq
See *torch.unbind()*

unfold(*dim*, *size*, *step*) → Tensor
Returns a tensor which contains all slices of size *size* from `self` tensor in the dimension *dim*.

Step between two slices is given by `step`.

If *sizedim* is the size of dimension *dim* for `self`, 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.

Parameters

- **dim** (*int*) – dimension in which unfolding happens
- **size** (*int*) – the size of each slice that is unfolded
- **step** (*int*) – the step between each slice

Example:

```
>>> x = torch.arange(1., 8)
>>> x
tensor([ 1.,  2.,  3.,  4.,  5.,  6.,  7.])
>>> x.unfold(0, 2, 1)
tensor([[ 1.,  2.],
        [ 2.,  3.],
        [ 3.,  4.],
        [ 4.,  5.],
        [ 5.,  6.],
        [ 6.,  7.]])
>>> x.unfold(0, 2, 2)
tensor([[ 1.,  2.],
        [ 3.,  4.],
        [ 5.,  6.]])
```

uniform_ (*from=0*, *to=1*) → Tensor
Fills `self` tensor with numbers sampled from the continuous uniform distribution:

$$P(x) = \frac{1}{\text{to} - \text{from}}$$

unique (*sorted=True*, *return_inverse=False*, *dim=None*)
Returns the unique scalar elements of the tensor as a 1-D tensor.

See *torch.unique()*

unsqueeze (*dim*) → Tensor
See *torch.unsqueeze()*

unsqueeze_ (*dim*) → Tensor
In-place version of *unsqueeze()*

values () → Tensor
If `self` is a sparse COO tensor (i.e., with `torch.sparse_coo` layout), this returns a view of the contained values tensor. Otherwise, this throws an error.

See also *Tensor.indices()*.

Note: This method can only be called on a coalesced sparse tensor. See `Tensor.coalesce()` for details.

---

var (*dim=None*, *unbiased=True*, *keepdim=False*) → Tensor
See *torch.var()*

view (**shape*) → Tensor
Returns a new tensor with the same data as the `self` tensor but of a different `shape`.

The returned tensor shares the same data and must have the same number of elements, but may have a different size. For a tensor to be viewed, the new view size must be compatible with its original size and stride, i.e., each new view dimension must either be a subspace of an original dimension, or only span across original dimensions $d, d+1, \ldots, d+k$ that satisfy the following contiguity-like condition that $\forall i = 0, \ldots, k-1$,

$$\text{stride}[i] = \text{stride}[i+1] \times \text{size}[i+1]$$

Otherwise, *contiguous()* needs to be called before the tensor can be viewed. See also: *reshape()*, which returns a view if the shapes are compatible, and copies (equivalent to calling *contiguous()*) otherwise.

Parameters **shape** (*torch.Size or int...*) – the 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])
```

(continues on next page)

(continued from previous page)

```
>>> a = torch.randn(1, 2, 3, 4)
>>> a.size()
torch.Size([1, 2, 3, 4])
>>> b = a.transpose(1, 2)  # Swaps 2nd and 3rd dimension
>>> b.size()
torch.Size([1, 3, 2, 4])
>>> c = a.view(1, 3, 2, 4)  # Does not change tensor layout in memory
>>> c.size()
torch.Size([1, 3, 2, 4])
>>> torch.equal(b, c)
False
```

view_as(*other*) → Tensor

View this tensor as the same size as `other`. `self.view_as(other)` is equivalent to `self.view(other.size())`.

Please see *view()* for more information about `view`.

Parameters **other** (*torch.Tensor*) – The result tensor has the same size as `other`.

where(*condition*, *y*) → Tensor

`self.where(condition, y)` is equivalent to `torch.where(condition, self, y)`. See *torch.where()*

zero_() → Tensor

Fills `self` tensor with zeros.

class torch.**BoolTensor**

The following methods are unique to *torch.BoolTensor*.

all()

all() → bool

Returns True if all elements in the tensor are non-zero, False otherwise.

Example:

```
>>> a = torch.randn(1, 3).byte() % 2
>>> a
tensor([[1, 0, 0]], dtype=torch.uint8)
>>> a.all()
tensor(0, dtype=torch.uint8)
```

all(*dim*, *keepdim=False*, *out=None*) → Tensor

Returns True if all elements in each row of the tensor in the given dimension `dim` are non-zero, False otherwise.

If `keepdim` is `True`, the output tensor is of the same size as `input` except in the dimension `dim` where it is of size 1. Otherwise, `dim` is squeezed (see *torch.squeeze()*), resulting in the output tensor having 1 fewer dimension than `input`.

Parameters

- **dim** (*int*) – the dimension to reduce
- **keepdim** (*bool*) – whether the output tensor has `dim` retained or not
- **out** (*Tensor, optional*) – the output tensor

Example:

```
>>> a = torch.randn(4, 2).byte() % 2
>>> a
tensor([[0, 0],
        [0, 0],
        [0, 1],
        [1, 1]], dtype=torch.uint8)
>>> a.all(dim=1)
tensor([0, 0, 0, 1], dtype=torch.uint8)
```

any()

any() → bool

Returns True if any elements in the tensor are non-zero, False otherwise.

Example:

```
>>> a = torch.randn(1, 3).byte() % 2
>>> a
tensor([[0, 0, 1]], dtype=torch.uint8)
>>> a.any()
tensor(1, dtype=torch.uint8)
```

any(*dim*, *keepdim=False*, *out=None*) → Tensor

Returns True if any elements in each row of the tensor in the given dimension `dim` are non-zero, False otherwise.

If `keepdim` is `True`, the output tensor is of the same size as `input` except in the dimension `dim` where it is of size 1. Otherwise, `dim` is squeezed (see *`torch.squeeze()`*), resulting in the output tensor having 1 fewer dimension than `input`.

Parameters

- **dim** (*int*) – the dimension to reduce
- **keepdim** (*bool*) – whether the output tensor has `dim` retained or not
- **out** (`Tensor`, *optional*) – the output tensor

Example:

```
>>> a = torch.randn(4, 2).byte() % 2
>>> a
tensor([[1, 0],
        [0, 0],
        [0, 1],
        [0, 0]], dtype=torch.uint8)
>>> a.any(dim=1)
tensor([1, 0, 1, 0], dtype=torch.uint8)
```

CHAPTER

NINETEEN

TENSOR ATTRIBUTES

Each `torch.Tensor` has a *torch.dtype*, *torch.device*, and *torch.layout*.

19.1 torch.dtype

class torch.**dtype**

A *torch.dtype* is an object that represents the data type of a *torch.Tensor*. PyTorch has nine different data types:

Data type	dtype	Tensor types
32-bit floating point	`torch.float32` or `torch.float`	`torch.*.FloatTensor`
64-bit floating point	`torch.float64` or `torch.double`	`torch.*.DoubleTensor`
16-bit floating point	`torch.float16` or `torch.half`	`torch.*.HalfTensor`
8-bit integer (unsigned)	`torch.uint8`	`torch.*.ByteTensor`
8-bit integer (signed)	`torch.int8`	`torch.*.CharTensor`
16-bit integer (signed)	`torch.int16` or `torch.short`	`torch.*.ShortTensor`
32-bit integer (signed)	`torch.int32` or `torch.int`	`torch.*.IntTensor`
64-bit integer (signed)	`torch.int64` or `torch.long`	`torch.*.LongTensor`
Boolean	`torch.bool`	`torch.*.BoolTensor`

To find out if a *torch.dtype* is a floating point data type, the property *is_floating_point* can be used, which returns `True` if the data type is a floating point data type.

When the dtypes of inputs to an arithmetic operation (*add*, *sub*, *div*, *mul*) differ, we promote by finding the minimum dtype that satisfies the following rules:

- If the type of a scalar operand is of a higher category than tensor operands (where floating > integral > boolean), we promote to a type with sufficient size to hold all scalar operands of that category.
- If a zero-dimension tensor operand has a higher category than dimensioned operands, we promote to a type with sufficient size and category to hold all zero-dim tensor operands of that category.
- If there are no higher-category zero-dim operands, we promote to a type with sufficient size and category to hold all dimensioned operands.

A floating point scalar operand has dtype *torch.get_default_dtype()* and an integral non-boolean scalar operand has dtype *torch.int64*. Unlike numpy, we do not inspect values when determining the minimum *dtypes* of an operand. Quantized and complex types are not yet supported.

Promotion Examples:

```
>>> float_tensor = torch.ones(1, dtype=torch.float)
>>> double_tensor = torch.ones(1, dtype=torch.double)
>>> int_tensor = torch.ones(1, dtype=torch.int)
>>> long_tensor = torch.ones(1, dtype=torch.long)
>>> uint_tensor = torch.ones(1, dtype=torch.uint8)
>>> double_tensor = torch.ones(1, dtype=torch.double)
>>> bool_tensor = torch.ones(1, dtype=torch.bool)
# zero-dim tensors
>>> long_zerodim = torch.tensor(1, dtype=torch.long)
>>> int_zerodim = torch.tensor(1, dtype=torch.int)

>>> torch.add(5, 5).dtype
torch.int64
# 5 is an int64, but does not have higher category than int_tensor so is not␣
↪considered.
>>> (int_tensor + 5).dtype
torch.int32
>>> (int_tensor + long_zerodim).dtype
torch.int32
>>> (long_tensor + int_tensor).dtype
torch.int64
>>> (bool_tensor + long_tensor).dtype
torch.int64
>>> (bool_tensor + uint_tensor).dtype
torch.uint8
>>> (float_tensor + double_tensor).dtype
torch.float64
>>> (bool_tensor + int_tensor).dtype
torch.int32
# Since long is a different kind than float, result dtype only needs to be large␣
↪enough
# to hold the float.
>>> torch.add(long_tensor, float_tensor).dtype
torch.float32
```

When the output tensor of an arithmetic operation is specified, we allow casting to its *dtype* except that:

- An integral output tensor cannot accept a floating point tensor.
- A boolean output tensor cannot accept a non-boolean tensor.

Casting Examples:

```
# allowed:
>>> float_tensor *= double_tensor
>>> float_tensor *= int_tensor
>>> float_tensor *= uint_tensor
>>> float_tensor *= bool_tensor
>>> float_tensor *= double_tensor
>>> int_tensor *= long_tensor
>>> int_tensor *= uint_tensor
>>> uint_tensor *= int_tensor

# disallowed (RuntimeError: result type can't be cast to the desired output type):
>>> int_tensor *= float_tensor
>>> bool_tensor *= int_tensor
>>> bool_tensor *= uint_tensor
```

19.2 torch.device

class torch.**device**

A *torch.device* is an object representing the device on which a *torch.Tensor* is or will be allocated.

The *torch.device* contains a device type (`'cpu'` or `'cuda'`) and optional device ordinal for the device type. If the device ordinal is not present, this object will always represent the current device for the device type, even after *torch.cuda.set_device()* is called; e.g., a *torch.Tensor* constructed with device `'cuda'` is equivalent to `'cuda:X'` where X is the result of *torch.cuda.current_device()*.

A *torch.Tensor*s device can be accessed via the *Tensor.device* property.

A *torch.device* can be constructed via a string or via a string and device ordinal

Via a string:

```
>>> torch.device('cuda:0')
device(type='cuda', index=0)

>>> torch.device('cpu')
device(type='cpu')

>>> torch.device('cuda')  # current cuda device
device(type='cuda')
```

Via a string and device ordinal:

```
>>> torch.device('cuda', 0)
device(type='cuda', index=0)

>>> torch.device('cpu', 0)
device(type='cpu', index=0)
```

Note: The *torch.device* argument in functions can generally be substituted with a string. This allows for fast prototyping of code.

```
>>> # Example of a function that takes in a torch.device
>>> cuda1 = torch.device('cuda:1')
>>> torch.randn((2,3), device=cuda1)
```

```
>>> # You can substitute the torch.device with a string
>>> torch.randn((2,3), device='cuda:1')
```

Note: For legacy reasons, a device can be constructed via a single device ordinal, which is treated as a cuda device. This matches *Tensor.get_device()*, which returns an ordinal for cuda tensors and is not supported for cpu tensors.

```
>>> torch.device(1)
device(type='cuda', index=1)
```

Note: Methods which take a device will generally accept a (properly formatted) string or (legacy) integer device ordinal, i.e. the following are all equivalent:

```
>>> torch.randn((2,3), device=torch.device('cuda:1'))
>>> torch.randn((2,3), device='cuda:1')
>>> torch.randn((2,3), device=1)  # legacy
```

19.3 torch.layout

class torch.**layout**

A *torch.layout* is an object that represents the memory layout of a *torch.Tensor*. Currently, we support `torch.strided` (dense Tensors) and have experimental support for `torch.sparse_coo` (sparse COO Tensors).

`torch.strided` represents dense Tensors and is the memory layout that is most commonly used. Each strided tensor has an associated `torch.Storage`, which holds its data. These tensors provide multi-dimensional, strided view of a storage. Strides are a list of integers: the k-th stride represents the jump in the memory necessary to go from one element to the next one in the k-th dimension of the Tensor. This concept makes it possible to perform many tensor operations efficiently.

Example:

```
>>> x = torch.Tensor([[1, 2, 3, 4, 5], [6, 7, 8, 9, 10]])
>>> x.stride()
(5, 1)

>>> x.t().stride()
(1, 5)
```

For more information on `torch.sparse_coo` tensors, see *torch.sparse*.

CHAPTER

TWENTY

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 declare `Tensor` s for which gradients should be computed with the `requires_grad=True` keyword.

`torch.autograd.`**`backward`**(*tensors*, *grad_tensors=None*, *retain_graph=None*, *create_graph=False*, *grad_variables=None*)

Computes the sum of gradients of given tensors w.r.t. graph leaves.

The graph is differentiated using the chain rule. If any of `tensors` are non-scalar (i.e. their data has more than one element) and require gradient, then the Jacobian-vector product would be computed, in this case the function additionally requires specifying `grad_tensors`. It should be a sequence of matching length, that contains the vector in the Jacobian-vector product, usually the gradient of the differentiated function w.r.t. corresponding tensors (`None` is an acceptable value for all tensors that dont need gradient tensors).

This function accumulates gradients in the leaves - you might need to zero them before calling it.

Parameters

- **`tensors`** (*sequence of Tensor*) – Tensors of which the derivative will be computed.
- **`grad_tensors`** (*sequence of (Tensor or None)*) – The vector in the Jacobian-vector product, usually gradients w.r.t. each element of corresponding tensors. None values can be specified for scalar Tensors or ones that dont require grad. If a None value would be acceptable for all grad_tensors, then this argument is optional.
- **`retain_graph`** (*bool, optional*) – If `False`, the graph used to compute the grad will be freed. Note that in nearly all cases setting this option to `True` is not needed and often can be worked around in a much more efficient way. Defaults to the value of `create_graph`.
- **`create_graph`** (*bool, optional*) – If `True`, graph of the derivative will be constructed, allowing to compute higher order derivative products. Defaults to `False`.

`torch.autograd.`**`grad`**(*outputs*, *inputs*, *grad_outputs=None*, *retain_graph=None*, *create_graph=False*, *only_inputs=True*, *allow_unused=False*)

Computes and returns the sum of gradients of outputs w.r.t. the inputs.

`grad_outputs` should be a sequence of length matching `output` containing the vector in Jacobian-vector product, usually the pre-computed gradients w.r.t. each of the outputs. If an output doesnt require_grad, then the gradient can be `None`).

If `only_inputs` is `True`, the function will only return a list of gradients w.r.t the specified inputs. If its `False`, then gradient w.r.t. all remaining leaves will still be computed, and will be accumulated into their `.grad` attribute.

Parameters

- **`outputs`** (*sequence of Tensor*) – outputs of the differentiated function.

- **inputs** (*sequence of Tensor*) – Inputs w.r.t. which the gradient will be returned (and not accumulated into `.grad`).
- **grad_outputs** (*sequence of Tensor*) – The vector in the Jacobian-vector product. Usually gradients w.r.t. each output. None values can be specified for scalar Tensors or ones that dont require grad. If a None value would be acceptable for all grad_tensors, then this argument is optional. Default: None.
- **retain_graph** (*bool, optional*) – If `False`, the graph used to compute the grad will be freed. Note that in nearly all cases setting this option to `True` is not needed and often can be worked around in a much more efficient way. Defaults to the value of `create_graph`.
- **create_graph** (*bool, optional*) – If `True`, graph of the derivative will be constructed, allowing to compute higher order derivative products. Default: `False`.
- **allow_unused** (*bool, optional*) – If `False`, specifying inputs that were not used when computing outputs (and therefore their grad is always zero) is an error. Defaults to `False`.

20.1 Locally disabling gradient computation

class torch.autograd.**no_grad**

Context-manager that disabled gradient calculation.

Disabling gradient calculation is useful for inference, when you are sure that you will not call `Tensor.backward()`. It will reduce memory consumption for computations that would otherwise have *requires_grad=True*. In this mode, the result of every computation will have *requires_grad=False*, even when the inputs have *requires_grad=True*.

Also functions as a decorator.

Example:

```
>>> x = torch.tensor([1], requires_grad=True)
>>> with torch.no_grad():
...   y = x * 2
>>> y.requires_grad
False
>>> @torch.no_grad()
... def doubler(x):
...     return x * 2
>>> z = doubler(x)
>>> z.requires_grad
False
```

class torch.autograd.**enable_grad**

Context-manager that enables gradient calculation.

Enables gradient calculation inside a *no_grad* context. This has no effect outside of *no_grad*.

Also functions as a decorator.

Example:

```
>>> x = torch.tensor([1], requires_grad=True)
>>> with torch.no_grad():
...   with torch.enable_grad():
```

(continues on next page)

(continued from previous page)

```
...     y = x * 2
>>> y.requires_grad
True
>>> y.backward()
>>> x.grad
>>> @torch.enable_grad()
... def doubler(x):
...     return x * 2
>>> with torch.no_grad():
...     z = doubler(x)
>>> z.requires_grad
True
```

class torch.autograd.**set_grad_enabled**(*mode*)

Context-manager that sets gradient calculation to on or off.

`set_grad_enabled` will enable or disable grads based on its argument `mode`. It can be used as a context-manager or as a function.

Parameters **mode** (*bool*) – Flag whether to enable grad (`True`), or disable (`False`). This can be used to conditionally enable gradients.

Example:

```
>>> x = torch.tensor([1], requires_grad=True)
>>> is_train = False
>>> with torch.set_grad_enabled(is_train):
...   y = x * 2
>>> y.requires_grad
False
>>> torch.set_grad_enabled(True)
>>> y = x * 2
>>> y.requires_grad
True
>>> torch.set_grad_enabled(False)
>>> y = x * 2
>>> y.requires_grad
False
```

20.2 In-place operations on Tensors

Supporting in-place operations in autograd is a hard matter, and we discourage their use in most cases. Autograds 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 youre operating under heavy memory pressure, you might never need to use them.

20.2.1 In-place correctness checks

All `Tensor` s keep track of in-place operations applied to them, and if the implementation detects that a tensor 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 youre using in-place functions and not seeing any errors, you can be sure that the computed gradients are correct.

20.3 Variable (deprecated)

Warning: The Variable API has been deprecated: Variables are no longer necessary to use autograd with tensors. Autograd automatically supports Tensors with `requires_grad` set to `True`. Below please find a quick guide on what has changed:

- `Variable(tensor)` and `Variable(tensor, requires_grad)` still work as expected, but they return Tensors instead of Variables.
- `var.data` is the same thing as `tensor.data`.
- Methods such as `var.backward(), var.detach(), var.register_hook()` now work on tensors with the same method names.

In addition, one can now create tensors with `requires_grad=True` using factory methods such as *`torch.randn()`*, *`torch.zeros()`*, *`torch.ones()`*, and others like the following:

```
autograd_tensor = torch.randn((2, 3, 4), requires_grad=True)
```

20.4 Tensor autograd functions

class torch.**Tensor**

grad
This attribute is `None` by default and becomes a Tensor the first time a call to *`backward()`* computes gradients for `self`. The attribute will then contain the gradients computed and future calls to *`backward()`* will accumulate (add) gradients into it.

requires_grad
Is `True` if gradients need to be computed for this Tensor, `False` otherwise.

Note: The fact that gradients need to be computed for a Tensor do not mean that the *`grad`* attribute will be populated, see *`is_leaf`* for more details.

---

is_leaf
All Tensors that have *`requires_grad`* which is `False` will be leaf Tensors by convention.

For Tensors that have *`requires_grad`* which is `True`, they will be leaf Tensors if they were created by the user. This means that they are not the result of an operation and so `grad_fn` is None.

Only leaf Tensors will have their *`grad`* populated during a call to *`backward()`*. To get *`grad`* populated for non-leaf Tensors, you can use *`retain_grad()`*.

Example:

```
>>> a = torch.rand(10, requires_grad=True)
>>> a.is_leaf
True
>>> b = torch.rand(10, requires_grad=True).cuda()
>>> b.is_leaf
False
# b was created by the operation that cast a cpu Tensor into a cuda Tensor
>>> c = torch.rand(10, requires_grad=True) + 2
```

(continues on next page)

(continued from previous page)

```
>>> c.is_leaf
False
# c was created by the addition operation
>>> d = torch.rand(10).cuda()
>>> d.is_leaf
True
# d does not require gradients and so has no operation creating it (that is␣
↪tracked by the autograd engine)
>>> e = torch.rand(10).cuda().requires_grad_()
>>> e.is_leaf
True
# e requires gradients and has no operations creating it
>>> f = torch.rand(10, requires_grad=True, device="cuda")
>>> f.is_leaf
True
# f requires grad, has no operation creating it
```

backward(*gradient=None*, *retain_graph=None*, *create_graph=False*)

Computes the gradient of current tensor w.r.t. graph leaves.

The graph is differentiated using the chain rule. If the tensor is non-scalar (i.e. its data has more than one element) and requires gradient, the function additionally 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.

Parameters

- **gradient** (*Tensor or None*) – Gradient w.r.t. the tensor. If it is a tensor, it will be automatically converted to a Tensor that does not require grad unless `create_graph` is True. None values can be specified for scalar Tensors or ones that dont require grad. If a None value would be acceptable then this argument is optional.
- **retain_graph** (*bool, optional*) – If `False`, the graph used to compute the grads will be freed. Note that in nearly all cases setting this option to True is not needed and often can be worked around in a much more efficient way. Defaults to the value of `create_graph`.
- **create_graph** (*bool, optional*) – If `True`, graph of the derivative will be constructed, allowing to compute higher order derivative products. Defaults to `False`.

detach()

Returns a new Tensor, detached from the current graph.

The result will never require gradient.

Note: Returned Tensor shares the same storage with the original one. In-place modifications on either of them will be seen, and may trigger errors in correctness checks. IMPORTANT NOTE: Previously, in-place size / stride / storage changes (such as *resize_* / *resize_as_* / *set_* / *transpose_*) to the returned tensor also update the original tensor. Now, these in-place changes will not update the original tensor anymore, and will instead trigger an error. For sparse tensors: In-place indices / values changes (such as *zero_* / *copy_* / *add_*) to the returned tensor will not update the original tensor anymore, and will instead trigger an error.

---

detach_()

Detaches the Tensor from the graph that created it, making it a leaf. Views cannot be detached in-place.

register_hook(*hook*)
Registers a backward hook.

The hook will be called every time a gradient with respect to the Tensor is computed. The hook should have the following signature:

```
hook(grad) -> Tensor 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 = torch.tensor([0., 0., 0.], requires_grad=True)
>>> h = v.register_hook(lambda grad: grad * 2)  # double the gradient
>>> v.backward(torch.tensor([1., 2., 3.]))
>>> v.grad

 2
 4
 6
[torch.FloatTensor of size (3,)]

>>> h.remove()  # removes the hook
```

retain_grad()
Enables .grad attribute for non-leaf Tensors.

20.5 Function

class torch.autograd.**Function**
Records operation history and defines formulas for differentiating ops.

Every operation performed on `Tensor` 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.

Each function object is meant to be used only once (in the forward pass).

Examples:

```
>>> class Exp(Function):
>>>
>>>     @staticmethod
>>>     def forward(ctx, i):
>>>         result = i.exp()
>>>         ctx.save_for_backward(result)
>>>         return result
>>>
```

(continues on next page)

(continued from previous page)

```
>>>     @staticmethod
>>>     def backward(ctx, grad_output):
>>>         result, = ctx.saved_tensors
>>>         return grad_output * result
```

static backward(*ctx*, **grad_outputs*)

Defines a formula for differentiating the operation.

This function is to be overridden by all subclasses.

It must accept a context `ctx` as the first argument, followed by 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.

The context can be used to retrieve tensors saved during the forward pass. It also has an attribute `ctx.needs_input_grad` as a tuple of booleans representing whether each input needs gradient. E.g., *backward()* will have `ctx.needs_input_grad[0] = True` if the first input to *forward()* needs gradient computated w.r.t. the output.

static forward(*ctx*, **args*, ***kwargs*)

Performs the operation.

This function is to be overridden by all subclasses.

It must accept a context ctx as the first argument, followed by any number of arguments (tensors or other types).

The context can be used to store tensors that can be then retrieved during the backward pass.

20.6 Numerical gradient checking

torch.autograd.**gradcheck**(*func*, *inputs*, *eps=1e-06*, *atol=1e-05*, *rtol=0.001*, *raise_exception=True*, *check_sparse_nnz=False*)

Check gradients computed via small finite differences against analytical gradients w.r.t. tensors in `inputs` that are of floating point type and with `requires_grad=True`.

The check between numerical and analytical gradients uses *allclose()*.

Note: The default values are designed for `input` of double precision. This check will likely fail if `input` is of less precision, e.g., `FloatTensor`.

Warning: If any checked tensor in `input` has overlapping memory, i.e., different indices pointing to the same memory address (e.g., from `torch.expand()`), this check will likely fail because the numerical gradients computed by point perturbation at such indices will change values at all other indices that share the same memory address.

Parameters

- **func** (*function*) – a Python function that takes Tensor inputs and returns a Tensor or a tuple of Tensors
- **inputs** (*tuple of Tensor or Tensor*) – inputs to the function
- **eps** (*float, optional*) – perturbation for finite differences

- **atol** (*float, optional*) – absolute tolerance
- **rtol** (*float, optional*) – relative tolerance
- **raise_exception** (*bool, optional*) – indicating whether to raise an exception if the check fails. The exception gives more information about the exact nature of the failure. This is helpful when debugging gradchecks.
- **check_sparse_nnz** (*bool, optional*) – if True, gradcheck allows for SparseTensor input, and for any SparseTensor at input, gradcheck will perform check at nnz positions only.

Returns True if all differences satisfy allclose condition

torch.autograd.**gradgradcheck**(*func*, *inputs*, *grad_outputs=None*, *eps=1e-06*, *atol=1e-05*, *rtol=0.001*, *gen_non_contig_grad_outputs=False*, *raise_exception=True*)

Check gradients of gradients computed via small finite differences against analytical gradients w.r.t. tensors in `inputs` and `grad_outputs` that are of floating point type and with `requires_grad=True`.

This function checks that backpropagating through the gradients computed to the given `grad_outputs` are correct.

The check between numerical and analytical gradients uses *allclose()*.

Note: The default values are designed for `input` and `grad_outputs` of double precision. This check will likely fail if they are of less precision, e.g., `FloatTensor`.

Warning: If any checked tensor in `input` and `grad_outputs` has overlapping memory, i.e., different indices pointing to the same memory address (e.g., from `torch.expand()`), this check will likely fail because the numerical gradients computed by point perturbation at such indices will change values at all other indices that share the same memory address.

Parameters

- **func** (*function*) – a Python function that takes Tensor inputs and returns a Tensor or a tuple of Tensors
- **inputs** (*tuple of Tensor or Tensor*) – inputs to the function
- **grad_outputs** (*tuple of Tensor or Tensor, optional*) – The gradients with respect to the functions outputs.
- **eps** (*float, optional*) – perturbation for finite differences
- **atol** (*float, optional*) – absolute tolerance
- **rtol** (*float, optional*) – relative tolerance
- **gen_non_contig_grad_outputs** (*bool, optional*) – if `grad_outputs` is `None` and `gen_non_contig_grad_outputs` is `True`, the randomly generated gradient outputs are made to be noncontiguous
- **raise_exception** (*bool, optional*) – indicating whether to raise an exception if the check fails. The exception gives more information about the exact nature of the failure. This is helpful when debugging gradchecks.

Returns True if all differences satisfy allclose condition

20.7 Profiler

Autograd includes a profiler that lets you inspect the cost of different operators inside your model - both on the CPU and GPU. There are two modes implemented at the moment - CPU-only using *profile*. and nvprof based (registers both CPU and GPU activity) using *emit_nvtx*.

class torch.autograd.profiler.**profile**(*enabled=True*, *use_cuda=False*)

Context manager that manages autograd profiler state and holds a summary of results.

Parameters

- **enabled** (*bool, optional*) – Setting this to False makes this context manager a no-op. Default: `True`.
- **use_cuda** (*bool, optional*) – Enables timing of CUDA events as well using the cudaEvent API. Adds approximately 4us of overhead to each tensor operation. Default: `False`

Example

```
>>> x = torch.randn((1, 1), requires_grad=True)
>>> with torch.autograd.profiler.profile() as prof:
...     y = x ** 2
...     y.backward()
>>> # NOTE: some columns were removed for brevity
... print(prof)
-----------------------------------  ---------------  ---------------
Name                                          CPU time        CUDA time
-----------------------------------  ---------------  ---------------
PowConstant                                 142.036us          0.000us
N5torch8autograd9GraphRootE                  63.524us          0.000us
PowConstantBackward                         184.228us          0.000us
MulConstant                                  50.288us          0.000us
PowConstant                                  28.439us          0.000us
Mul                                          20.154us          0.000us
N5torch8autograd14AccumulateGradE            13.790us          0.000us
N5torch8autograd5CloneE                       4.088us          0.000us
```

export_chrome_trace(*path*)

Exports an EventList as a Chrome tracing tools file.

The checkpoint can be later loaded and inspected under `chrome://tracing` URL.

Parameters **path** (*str*) – Path where the trace will be written.

key_averages()

Averages all function events over their keys.

Returns An EventList containing FunctionEventAvg objects.

table(*sort_by=None*)

Prints an EventList as a nicely formatted table.

Parameters **sort_by** (*str, optional*) – Attribute used to sort entries. By default they are printed in the same order as they were registered. Valid keys include: `cpu_time`, `cuda_time`, `cpu_time_total`, `cuda_time_total`, `count`.

Returns A string containing the table.

total_average()
Averages all events.

Returns A FunctionEventAvg object.

class torch.autograd.profiler.**emit_nvtx**(*enabled=True*)
Context manager that makes every autograd operation emit an NVTX range.

It is useful when running the program under nvprof:

```
nvprof --profile-from-start off -o trace_name.prof -- <regular command here>
```

Unfortunately, theres no way to force nvprof to flush the data it collected to disk, so for CUDA profiling one has to use this context manager to annotate nvprof traces and wait for the process to exit before inspecting them. Then, either NVIDIA Visual Profiler (nvvp) can be used to visualize the timeline, or *torch.autograd.profiler.load_nvprof()* can load the results for inspection e.g. in Python REPL.

Parameters **enabled** (*bool, optional*) – Setting this to False makes this context manager a no-op. Default: True.

Example

```
>>> with torch.cuda.profiler.profile():
...     model(x) # Warmup CUDA memory allocator and profiler
...     with torch.autograd.profiler.emit_nvtx():
...         model(x)
```

Forward-backward correlation

When viewing a profile created using *emit_nvtx* in the Nvidia Visual Profiler, correlating each backward-pass op with the corresponding forward-pass op can be difficult. To ease this task, *emit_nvtx* appends sequence number information to the ranges it generates.

During the forward pass, each function range is decorated with seq=<N>. seq is a running counter, incremented each time a new backward Function object is created and stashed for backward. Thus, the *seq=<N>* annotation associated with each forward function range tells you that if a backward Function object is created by this forward function, the backward object will receive sequence number N. During the backward pass, the top-level range wrapping each C++ backward Functions apply() call is decorated with stashed seq=<M>. M is the sequence number that the backward object was created with. By comparing stashed seq numbers in backward with seq numbers in forward, you can track down which forward op created each backward Function.

Any functions executed during the backward pass are also decorated with seq=<N>. During default backward (with create_graph=False) this information is irrelevant, and in fact, N may simply be 0 for all such functions. Only the top-level ranges associated with backward Function objects apply() methods are useful, as a way to correlate these Function objects with the earlier forward pass.

Double-backward

If, on the other hand, a backward pass with create_graph=True is underway (in other words, if you are setting up for a double-backward), each functions execution during backward is given a nonzero, useful seq=<N>. Those functions may themselves create Function objects to be executed later during double-backward, just as the original functions in the forward pass did. The relationship between backward and double-backward is conceptually the same as the relationship between forward and backward: The functions still emit current-sequence-number-tagged ranges, the Function objects they create still stash those sequence numbers, and during the eventual double-backward, the Function objects apply() ranges are still tagged with stashed seq numbers, which can be compared to *seq* numbers from the backward pass.

torch.autograd.profiler.**load_nvprof**(*path*)
Opens an nvprof trace file and parses autograd annotations.

Parameters **path** (*str*) – path to nvprof trace

20.8 Anomaly detection

class torch.autograd.**detect_anomaly**
Context-manager that enable anomaly detection for the autograd engine.

This does two things: - Running the forward pass with detection enabled will allow the backward pass to print the traceback of the forward operation that created the failing backward function. - Any backward computation that generate nan value will raise an error.

Example

```
>>> import torch
>>> from torch import autograd
>>> class MyFunc(autograd.Function):
...     @staticmethod
...     def forward(ctx, inp):
...         return inp.clone()
...     @staticmethod
...     def backward(ctx, gO):
...         # Error during the backward pass
...         raise RuntimeError("Some error in backward")
...         return gO.clone()
>>> def run_fn(a):
...     out = MyFunc.apply(a)
...     return out.sum()
>>> inp = torch.rand(10, 10, requires_grad=True)
>>> out = run_fn(inp)
>>> out.backward()
    Traceback (most recent call last):
      File "<stdin>", line 1, in <module>
      File "/your/pytorch/install/torch/tensor.py", line 93, in backward
        torch.autograd.backward(self, gradient, retain_graph, create_graph)
      File "/your/pytorch/install/torch/autograd/__init__.py", line 90, in␣
↪backward
        allow_unreachable=True)  # allow_unreachable flag
      File "/your/pytorch/install/torch/autograd/function.py", line 76, in apply
        return self._forward_cls.backward(self, *args)
      File "<stdin>", line 8, in backward
    RuntimeError: Some error in backward
>>> with autograd.detect_anomaly():
...     inp = torch.rand(10, 10, requires_grad=True)
...     out = run_fn(inp)
...     out.backward()
    Traceback of forward call that caused the error:
      File "tmp.py", line 53, in <module>
        out = run_fn(inp)
      File "tmp.py", line 44, in run_fn
        out = MyFunc.apply(a)
    Traceback (most recent call last):
```

(continues on next page)

(continued from previous page)

```
      File "<stdin>", line 4, in <module>
      File "/your/pytorch/install/torch/tensor.py", line 93, in backward
        torch.autograd.backward(self, gradient, retain_graph, create_graph)
      File "/your/pytorch/install/torch/autograd/__init__.py", line 90, in␣
↪backward
        allow_unreachable=True)  # allow_unreachable flag
      File "/your/pytorch/install/torch/autograd/function.py", line 76, in apply
        return self._forward_cls.backward(self, *args)
      File "<stdin>", line 8, in backward
    RuntimeError: Some error in backward
```

class torch.autograd.**set_detect_anomaly**(*mode*)

Context-manager that sets the anomaly detection for the autograd engine on or off.

`set_detect_anomaly` will enable or disable the autograd anomaly detection based on its argument `mode`. It can be used as a context-manager or as a function.

See `detect_anomaly` above for details of the anomaly detection behaviour.

Parameters **mode** (*bool*) – Flag whether to enable anomaly detection (`True`), or disable (`False`).

CHAPTER

TWENTYONE

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**(*device=None*)
Returns the currently selected *Stream* for a given device.

Parameters **device** (*torch.device or int, optional*) – selected device. Returns the currently selected *Stream* for the current device, given by *current_device()*, if *device* is None (default).

torch.cuda.**default_stream**(*device=None*)
Returns the default *Stream* for a given device.

Parameters **device** (*torch.device or int, optional*) – selected device. Returns the default *Stream* for the current device, given by *current_device()*, if *device* is None (default).

class torch.cuda.**device**(*device*)
Context-manager that changes the selected device.

Parameters **device** (*torch.device or int*) – device index to select. Its a no-op if this argument is a negative integer or None.

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.

Parameters **obj** (*Tensor or Storage*) – object allocated on the selected device.

torch.cuda.**empty_cache**()
Releases all unoccupied cached memory currently held by the caching allocator so that those can be used in other GPU application and visible in *nvidia-smi*.

Note: *empty_cache()* doesnt increase the amount of GPU memory available for PyTorch. See *Memory management* for more details about GPU memory management.

torch.cuda.**get_device_capability**(*device=None*)
Gets the cuda capability of a device.

Parameters **device** (*torch.device or int, optional*) – device for which to return the device capability. This function is a no-op if this argument is a negative integer. Uses the current device, given by *current_device()*, if *device* is None (default).

Returns the major and minor cuda capability of the device

Return type tuple(int, int)

torch.cuda.**get_device_name**(*device=None*)
Gets the name of a device.

Parameters **device** (*torch.device or int, optional*) – device for which to return the name. This function is a no-op if this argument is a negative integer. Uses the current device, given by *current_device()*, if *device* is None (default).

torch.cuda.**init**()
Initialize PyTorchs CUDA state. You may need to call this explicitly if you are interacting with PyTorch via its C API, as Python bindings for CUDA functionality will not be until this initialization takes place. Ordinary users should not need this, as all of PyTorchs CUDA methods automatically initialize CUDA state on-demand.

Does nothing if the CUDA state is already initialized.

torch.cuda.**ipc_collect**()
Force collects GPU memory after it has been released by CUDA IPC.

Note: Checks if any sent CUDA tensors could be cleaned from the memory. Force closes shared memory file used for reference counting if there is no active counters. Useful when the producer process stopped actively sending tensors and want to release unused memory.

torch.cuda.**is_available**()
Returns a bool indicating if CUDA is currently available.

torch.cuda.**max_memory_allocated**(*device=None*)
Returns the maximum GPU memory occupied by tensors in bytes for a given device.

By default, this returns the peak allocated memory since the beginning of this program. *reset_max_memory_allocated()* can be used to reset the starting point in tracking this metric. For example, these two functions can measure the peak allocated memory usage of each iteration in a training loop.

Parameters **device** (*torch.device or int, optional*) – selected device. Returns statistic for the current device, given by *current_device()*, if *device* is None (default).

Note: See *Memory management* for more details about GPU memory management.

torch.cuda.**max_memory_cached**(*device=None*)
Returns the maximum GPU memory managed by the caching allocator in bytes for a given device.

By default, this returns the peak cached memory since the beginning of this program. *reset_max_memory_cached()* can be used to reset the starting point in tracking this metric. For

example, these two functions can measure the peak cached memory amount of each iteration in a training loop.

Parameters **device** (*torch.device or int, optional*) – selected device. Returns statistic for the current device, given by *current_device()*, if *device* is None (default).

Note: See *Memory management* for more details about GPU memory management.

torch.cuda.**memory_allocated**(*device=None*)

Returns the current GPU memory occupied by tensors in bytes for a given device.

Parameters **device** (*torch.device or int, optional*) – selected device. Returns statistic for the current device, given by *current_device()*, if *device* is None (default).

Note: This is likely less than the amount shown in *nvidia-smi* since some unused memory can be held by the caching allocator and some context needs to be created on GPU. See *Memory management* for more details about GPU memory management.

torch.cuda.**memory_cached**(*device=None*)

Returns the current GPU memory managed by the caching allocator in bytes for a given device.

Parameters **device** (*torch.device or int, optional*) – selected device. Returns statistic for the current device, given by *current_device()*, if *device* is None (default).

Note: See *Memory management* for more details about GPU memory management.

torch.cuda.**reset_max_memory_allocated**(*device=None*)

Resets the starting point in tracking maximum GPU memory occupied by tensors for a given device.

See *max_memory_allocated()* for details.

Parameters **device** (*torch.device or int, optional*) – selected device. Returns statistic for the current device, given by *current_device()*, if *device* is None (default).

Note: See *Memory management* for more details about GPU memory management.

torch.cuda.**reset_max_memory_cached**(*device=None*)

Resets the starting point in tracking maximum GPU memory managed by the caching allocator for a given device.

See *max_memory_cached()* for details.

Parameters **device** (*torch.device or int, optional*) – selected device. Returns statistic for the current device, given by *current_device()*, if *device* is None (default).

Note: See *Memory management* for more details about GPU memory management.

torch.cuda.**set_device**(*device*)

Sets the current device.

Usage of this function is discouraged in favor of *device*. In most cases its better to use CUDA_VISIBLE_DEVICES environmental variable.

Parameters **device** (*torch.device or int*) – selected device. This function is a no-op if this argument is negative.

torch.cuda.**stream**(*stream*)
Context-manager that selects a given stream.

All CUDA kernels queued within its context will be enqueued on a selected stream.

Parameters **stream** (*Stream*) – selected stream. This manager is a no-op if its None.

Note: Streams are per-device. If the selected stream is not on the current device, this function will also change the current device to match the stream.

---

torch.cuda.**synchronize**()
Waits for all kernels in all streams on current device to complete.

21.1 Random Number Generator

torch.cuda.**get_rng_state**(*device=device(type='cuda')*)
Returns the random number generator state of the current GPU as a ByteTensor.

Parameters **device** (*torch.device or int, optional*) – The device to return the RNG state of. Default: torch.device('cuda') (i.e., the current CUDA device).

Warning: This function eagerly initializes CUDA.

torch.cuda.**get_rng_state_all**()
Returns a tuple of ByteTensor representing the random number states of all devices.

torch.cuda.**set_rng_state**(*new_state*, *device=device(type='cuda')*)
Sets the random number generator state of the current GPU.

Parameters

- **new_state** (*torch.ByteTensor*) – The desired state
- **device** (*torch.device or int, optional*) – The device to set the RNG state. Default: torch.device('cuda') (i.e., the current CUDA device).

torch.cuda.**set_rng_state_all**(*new_states*)
Sets the random number generator state of all devices.

Parameters **new_state** (*tuple of torch.ByteTensor*) – The desired state for each device

torch.cuda.**manual_seed**(*seed*)
Sets the seed for generating random numbers for the current GPU. Its safe to call this function if CUDA is not available; in that case, it is silently ignored.

Parameters **seed** (*int*) – The desired seed.

Warning: If you are working with a multi-GPU model, this function is insufficient to get determinism. To seed all GPUs, use *manual_seed_all()*.

torch.cuda.**manual_seed_all**(*seed*)
Sets the seed for generating random numbers on all GPUs. Its safe to call this function if CUDA is not available; in that case, it is silently ignored.

Parameters **seed** (*int*) – The desired seed.

torch.cuda.**seed**()
Sets the seed for generating random numbers to a random number for the current GPU. Its safe to call this function if CUDA is not available; in that case, it is silently ignored.

Warning: If you are working with a multi-GPU model, this function will only initialize the seed on one GPU. To initialize all GPUs, use *seed_all()*.

torch.cuda.**seed_all**()
Sets the seed for generating random numbers to a random number on all GPUs. Its safe to call this function if CUDA is not available; in that case, it is silently ignored.

torch.cuda.**initial_seed**()
Returns the current random seed of the current GPU.

Warning: This function eagerly initializes CUDA.

21.2 Communication collectives

torch.cuda.comm.**broadcast**(*tensor*, *devices*)
Broadcasts a tensor to a number of GPUs.

Parameters

- **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.

Returns A tuple containing copies of the `tensor`, placed on devices corresponding to indices from `devices`.

torch.cuda.comm.**broadcast_coalesced**(*tensors*, *devices*, *buffer_size=10485760*)
Broadcasts a sequence tensors to the specified GPUs. Small tensors are first coalesced into a buffer to reduce the number of synchronizations.

Parameters

- **tensors** (*sequence*) – tensors 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.
- **buffer_size** (*int*) – maximum size of the buffer used for coalescing

Returns 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.

Parameters

- **inputs** (*Iterable[Tensor]*) – an iterable of tensors to add.
- **destination** (*int, optional*) – a device on which the output will be placed (default: current device).

Returns 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.

Parameters

- **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.

Returns A tuple containing chunks of the `tensor`, spread across 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.

Parameters

- **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)

Returns A tensor located on `destination` device, that is a result of concatenating `tensors` along `dim`.

21.3 Streams and events

class torch.cuda.**Stream**
Wrapper around a CUDA stream.

A CUDA stream is a linear sequence of execution that belongs to a specific device, independent from other streams. See *CUDA semantics* for details.

Parameters

- **device** (*torch.device or int, optional*) – a device on which to allocate the stream. If *device* is `None` (default) or a negative integer, this will use the current device.

- **priority** (*int, optional*) – priority of the stream. Lower numbers represent higher priorities.

query()
Checks if all the work submitted has been completed.

Returns A boolean indicating if all kernels in this stream are completed.

record_event(*event=None*)
Records an event.

Parameters **event** (Event, *optional*) – event to record. If not given, a new one will be allocated.

Returns Recorded event.

synchronize()
Wait for all the kernels in this stream to complete.

Note: This is a wrapper around `cudaStreamSynchronize()`: see **'CUDA documentation'_** for more info.

---

wait_event(*event*)
Makes all future work submitted to the stream wait for an event.

Parameters **event** (Event) – an event to wait for.

Note: This is a wrapper around `cudaStreamWaitEvent()`: see **'CUDA documentation'_** for more info.

This function returns without waiting for `event`: only future operations are affected.

---

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.

Parameters **stream** (Stream) – a stream to synchronize.

Note: This function returns without waiting for currently enqueued kernels in *stream*: only future operations are affected.

---

class torch.cuda.**Event**
Wrapper around a CUDA event.

CUDA events are synchronization markers that can be used to monitor the devices progress, to accurately measure timing, and to synchronize CUDA streams.

The underlying CUDA events are lazily initialized when the event is first recorded or exported to another process. After creation, only streams on the same device may record the event. However, streams on any device can wait on the event.

Parameters

- **enable_timing** (*bool, optional*) – indicates if the event should measure time (default: `False`)

- **blocking** (*bool, optional*) – if `True`, *wait()* will be blocking (default: `False`)
- **interprocess** () – if `True`, the event can be shared between processes (default: `False`)

elapsed_time(*end_event*)
Returns the time elapsed in milliseconds after the event was recorded and before the end_event was recorded.

classmethod from_ipc_handle(*device*, *handle*)
Reconstruct an event from an IPC handle on the given device.

ipc_handle()
Returns an IPC handle of this event. If not recorded yet, the event will use the current device.

query()
Checks if all work currently captured by event has completed.

Returns A boolean indicating if all work currently captured by event has completed.

record(*stream=None*)
Records the event in a given stream.

Uses `torch.cuda.current_stream()` if no stream is specified. The streams device must match the events device.

synchronize()
Waits for the event to complete.

Waits until the completion of all work currently captured in this event. This prevents the CPU thread from proceeding until the event completes.

Note: This is a wrapper around `cudaEventSynchronize()`: see **'CUDA documentation'_** for more info.

---

wait(*stream=None*)
Makes all future work submitted to the given stream wait for this event.

Use `torch.cuda.current_stream()` if no stream is specified.

21.4 Memory management

`torch.cuda.`**empty_cache**()
Releases all unoccupied cached memory currently held by the caching allocator so that those can be used in other GPU application and visible in *nvidia-smi*.

Note: *empty_cache()* doesnt increase the amount of GPU memory available for PyTorch. See *Memory management* for more details about GPU memory management.

`torch.cuda.`**memory_allocated**(*device=None*)
Returns the current GPU memory occupied by tensors in bytes for a given device.

Parameters **device** (*torch.device or int, optional*) – selected device. Returns statistic for the current device, given by *current_device()*, if *device* is `None` (default).

Note: This is likely less than the amount shown in *nvidia-smi* since some unused memory can be held by the caching allocator and some context needs to be created on GPU. See *Memory management* for more details about GPU memory management.

torch.cuda.**max_memory_allocated**(*device=None*)
Returns the maximum GPU memory occupied by tensors in bytes for a given device.

By default, this returns the peak allocated memory since the beginning of this program. *reset_max_memory_allocated()* can be used to reset the starting point in tracking this metric. For example, these two functions can measure the peak allocated memory usage of each iteration in a training loop.

Parameters **device** (*torch.device or int, optional*) – selected device. Returns statistic for the current device, given by *current_device()*, if *device* is None (default).

Note: See *Memory management* for more details about GPU memory management.

torch.cuda.**reset_max_memory_allocated**(*device=None*)
Resets the starting point in tracking maximum GPU memory occupied by tensors for a given device.

See *max_memory_allocated()* for details.

Parameters **device** (*torch.device or int, optional*) – selected device. Returns statistic for the current device, given by *current_device()*, if *device* is None (default).

Note: See *Memory management* for more details about GPU memory management.

torch.cuda.**memory_cached**(*device=None*)
Returns the current GPU memory managed by the caching allocator in bytes for a given device.

Parameters **device** (*torch.device or int, optional*) – selected device. Returns statistic for the current device, given by *current_device()*, if *device* is None (default).

Note: See *Memory management* for more details about GPU memory management.

torch.cuda.**max_memory_cached**(*device=None*)
Returns the maximum GPU memory managed by the caching allocator in bytes for a given device.

By default, this returns the peak cached memory since the beginning of this program. *reset_max_memory_cached()* can be used to reset the starting point in tracking this metric. For example, these two functions can measure the peak cached memory amount of each iteration in a training loop.

Parameters **device** (*torch.device or int, optional*) – selected device. Returns statistic for the current device, given by *current_device()*, if *device* is None (default).

Note: See *Memory management* for more details about GPU memory management.

torch.cuda.**reset_max_memory_cached**(*device=None*)
Resets the starting point in tracking maximum GPU memory managed by the caching allocator for a given device.

See *max_memory_cached()* for details.

Parameters **device** (*torch.device or int, optional*) – selected device. Returns statistic for the current device, given by *current_device()*, if *device* is None (default).

Note: See *Memory management* for more details about GPU memory management.

21.5 NVIDIA Tools Extension (NVTX)

torch.cuda.nvtx.**mark**(*msg*)
Describe an instantaneous event that occurred at some point.

Parameters **msg** (*string*) – ASCII message to associate with the event.

torch.cuda.nvtx.**range_push**(*msg*)
Pushes a range onto a stack of nested range span. Returns zero-based depth of the range that is started.

Parameters **msg** (*string*) – ASCII message to associate with range

torch.cuda.nvtx.**range_pop**()
Pops a range off of a stack of nested range spans. Returns the zero-based depth of the range that is ended.

CHAPTER

TWENTYTWO

DISTRIBUTED COMMUNICATION PACKAGE - TORCH.DISTRIBUTED

22.1 Backends

`torch.distributed` supports three backends, each with different capabilities. The table below shows which functions are available for use with CPU / CUDA tensors. MPI supports CUDA only if the implementation used to build PyTorch supports it.

Backend	`gloo`		`mpi`		`nccl`	
Device	CPU	GPU	CPU	GPU	CPU	GPU
send	✓		✓	?		
recv	✓		✓	?		
broadcast	✓	✓	✓	?		✓
all_reduce	✓	✓	✓	?		✓
reduce	✓		✓	?		✓
all_gather	✓		✓	?		✓
gather	✓		✓	?		
scatter	✓		✓	?		
barrier	✓		✓	?		✓

22.1.1 Backends that come with PyTorch

PyTorch distributed currently only supports Linux. By default, the Gloo and NCCL backends are built and included in PyTorch distributed (NCCL only when building with CUDA). MPI is an optional backend that can only be included if you build PyTorch from source. (e.g. building PyTorch on a host that has MPI installed.)

22.1.2 Which backend to use?

In the past, we were often asked: which backend should I use?.

- Rule of thumb
  - Use the NCCL backend for distributed **GPU** training
  - Use the Gloo backend for distributed **CPU** training.
- GPU hosts with InfiniBand interconnect
  - Use NCCL, since its the only backend that currently supports InfiniBand and GPUDirect.
- GPU hosts with Ethernet interconnect

  - Use NCCL, since it currently provides the best distributed GPU training performance, especially for multiprocess single-node or multi-node distributed training. If you encounter any problem with NCCL, use Gloo as the fallback option. (Note that Gloo currently runs slower than NCCL for GPUs.)

- CPU hosts with InfiniBand interconnect
  - If your InfiniBand has enabled IP over IB, use Gloo, otherwise, use MPI instead. We are planning on adding InfiniBand support for Gloo in the upcoming releases.
- CPU hosts with Ethernet interconnect
  - Use Gloo, unless you have specific reasons to use MPI.

22.1.3 Common environment variables

Choosing the network interface to use

By default, both the NCCL and Gloo backends will try to find the right network interface to use. If the automatically detected interface is not correct, you can override it using the following environment variables (applicable to the respective backend):

- **NCCL_SOCKET_IFNAME**, for example `export NCCL_SOCKET_IFNAME=eth0`
- **GLOO_SOCKET_IFNAME**, for example `export GLOO_SOCKET_IFNAME=eth0`

If youre using the Gloo backend, you can specify multiple interfaces by separating them by a comma, like this: `export GLOO_SOCKET_IFNAME=eth0,eth1,eth2,eth3`. The backend will dispatch operations in a round-robin fashion across these interfaces. It is imperative that all processes specify the same number of interfaces in this variable.

Other NCCL environment variables

NCCL has also provided a number of environment variables for fine-tuning purposes.

Commonly used ones include the following for debugging purposes:

- `export NCCL_DEBUG=INFO`
- `export NCCL_DEBUG_SUBSYS=ALL`

For the full list of NCCL environment variables, please refer to NVIDIA NCCLs official documentation

22.2 Basics

The *torch.distributed* package provides PyTorch support and communication primitives for multiprocess parallelism across several computation nodes running on one or more machines. The class *torch.nn.parallel.DistributedDataParallel()* builds on this functionality to provide synchronous distributed training as a wrapper around any PyTorch model. This differs from the kinds of parallelism provided by multiprocessing and *torch.nn.DataParallel()* in that it supports multiple network-connected machines and in that the user must explicitly launch a separate copy of the main training script for each process.

In the single-machine synchronous case, *torch.distributed* or the *torch.nn.parallel.DistributedDataParallel()* wrapper may still have advantages over other approaches to data-parallelism, including *torch.nn.DataParallel()*:

- Each process maintains its own optimizer and performs a complete optimization step with each iteration. While this may appear redundant, since the gradients have already been gathered together and averaged across processes and are thus the same for every process, this means that no parameter broadcast step is needed, reducing time spent transferring tensors between nodes.
- Each process contains an independent Python interpreter, eliminating the extra interpreter overhead and GIL-thrashing that comes from driving several execution threads, model replicas, or GPUs from a single Python process. This is especially important for models that make heavy use of the Python runtime, including models with recurrent layers or many small components.

22.3 Initialization

The package needs to be initialized using the *torch.distributed.init_process_group()* function before calling any other methods. This blocks until all processes have joined.

torch.distributed.**init_process_group**(*backend*, *init_method='env://'*, *timeout=datetime.timedelta(0, 1800)*, ***kwargs*)

Initializes the default distributed process group, and this will also initialize the distributed package

Parameters

- **backend** (*str or* Backend) – The backend to use. Depending on build-time configurations, valid values include `mpi`, `gloo`, and `nccl`. This field should be given as a lowercase string (e.g., `"gloo"`), which can also be accessed via *Backend* attributes (e.g., `Backend.GLOO`). If using multiple processes per machine with `nccl` backend, each process must have exclusive access to every GPU it uses, as sharing GPUs between processes can result in deadlocks.
- **init_method** (*str, optional*) – URL specifying how to initialize the process group.
- **world_size** (*int, optional*) – Number of processes participating in the job.
- **rank** (*int, optional*) – Rank of the current process.
- **store** (*Store, optional*) – Rendevous key/value store as an alternative to other init methods.
- **timeout** (*timedelta, optional*) – Timeout for operations executed against the process group. Default value equals 30 minutes. This is only applicable for the `gloo` backend.
- **group_name** (*str, optional, deprecated*) – Group name.

To enable `backend == Backend.MPI`, PyTorch needs to built from source on a system that supports MPI. The same applies to NCCL as well.

class torch.distributed.**Backend**

An enum-like class of available backends: GLOO, NCCL, and MPI.

The values of this class are lowercase strings, e.g., `"gloo"`. They can be accessed as attributes, e.g., `Backend.NCCL`.

This class can be directly called to parse the string, e.g., `Backend(backend_str)` will check if `backend_str` is valid, and return the parsed lowercase string if so. It also accepts uppercase strings, e.g., `Backend("GLOO")` returns `"gloo"`.

Note: The entry `Backend.UNDEFINED` is present but only used as initial value of some fields. Users should neither use it directly nor assume its existence.

---

torch.distributed.**get_backend**(*group=<object object>*)
Returns the backend of the given process group.

Parameters **group** (*ProcessGroup, optional*) – The process group to work on. The default is the general main process group. If another specific group is specified, the calling process must be part of `group`.

Returns The backend of the given process group as a lower case string.

torch.distributed.**get_rank**(*group=<object object>*)
Returns the rank of current process group

Rank is a unique identifier assigned to each process within a distributed process group. They are always consecutive integers ranging from 0 to `world_size`.

Parameters **group** (*ProcessGroup, optional*) – The process group to work on

Returns The rank of the process group -1, if not part of the group

torch.distributed.**get_world_size**(*group=<object object>*)
Returns the number of processes in the current process group

Parameters **group** (*ProcessGroup, optional*) – The process group to work on

Returns The world size of the process group -1, if not part of the group

torch.distributed.**is_initialized**()
Checking if the default process group has been initialized

torch.distributed.**is_mpi_available**()
Checks if MPI is available

torch.distributed.**is_nccl_available**()
Checks if NCCL is available

Currently three initialization methods are supported:

22.3.1 TCP initialization

There are two ways to initialize using TCP, both requiring a network address reachable from all processes and a desired `world_size`. The first way requires specifying an address that belongs to the rank 0 process. This initialization method requires that all processes have manually specified ranks.

Note that multicast address is not supported anymore in the latest distributed package. `group_name` is deprecated as well.

```
import torch.distributed as dist

# Use address of one of the machines
dist.init_process_group(backend, init_method='tcp://10.1.1.20:23456',
                        rank=args.rank, world_size=4)
```

22.3.2 Shared file-system initialization

Another initialization method makes use of a file system that is shared and visible from all machines in a group, along with a desired `world_size`. The URL should start with `file://` and contain a path to a non-existent file (in an existing directory) on a shared file system. File-system initialization will automatically create that file if it doesnt exist,

but will not delete the file. Therefore, it is your responsibility to make sure that the file is cleaned up before the next *init_process_group()* call on the same file path/name.

Note that automatic rank assignment is not supported anymore in the latest distributed package and group_name is deprecated as well.

Warning: This method assumes that the file system supports locking using fcntl - most local systems and NFS support it.

Warning: This method will always create the file and try its best to clean up and remove the file at the end of the program. In other words, each initialization with the file init method will need a brand new empty file in order for the initialization to succeed. If the same file used by the previous initialization (which happens not to get cleaned up) is used again, this is unexpected behavior and can often cause deadlocks and failures. Therefore, even though this method will try its best to clean up the file, if the auto-delete happens to be unsuccessful, it is your responsibility to ensure that the file is removed at the end of the training to prevent the same file to be reused again during the next time. This is especially important if you plan to call *init_process_group()* multiple times on the same file name. In other words, if the file is not removed/cleaned up and you call *init_process_group()* again on that file, failures are expected. The rule of thumb here is that, make sure that the file is non-existent or empty everytime *init_process_group()* is called.

```
import torch.distributed as dist

# rank should always be specified
dist.init_process_group(backend, init_method='file:///mnt/nfs/sharedfile',
                        world_size=4, rank=args.rank)
```

22.3.3 Environment variable initialization

This method will read the configuration from environment variables, allowing one to fully customize how the information is obtained. The variables to be set are:

- MASTER_PORT - required; has to be a free port on machine with rank 0
- MASTER_ADDR - required (except for rank 0); address of rank 0 node
- WORLD_SIZE - required; can be set either here, or in a call to init function
- RANK - required; can be set either here, or in a call to init function

The machine with rank 0 will be used to set up all connections.

This is the default method, meaning that init_method does not have to be specified (or can be env://).

22.4 Groups

By default collectives operate on the default group (also called the world) and require all processes to enter the distributed function call. However, some workloads can benefit from more fine-grained communication. This is where distributed groups come into play. *new_group()* function can be used to create new groups, with arbitrary subsets of all processes. It returns an opaque group handle that can be given as a group argument to all collectives (collectives are distributed functions to exchange information in certain well-known programming patterns).

torch.distributed.**new_group**(*ranks=None*, *timeout=datetime.timedelta(0, 1800)*)
Creates a new distributed group.

This function requires that all processes in the main group (i.e. all processes that are part of the distributed job) enter this function, even if they are not going to be members of the group. Additionally, groups should be created in the same order in all processes.

Parameters

- **ranks** (*list[int]*) – List of ranks of group members.
- **timeout** (*timedelta, optional*) – Timeout for operations executed against the process group. Default value equals 30 minutes. This is only applicable for the `gloo` backend.

Returns A handle of distributed group that can be given to collective calls.

22.5 Point-to-point communication

torch.distributed.**send**(*tensor*, *dst*, *group=<object object>*, *tag=0*)
Sends a tensor synchronously.

Parameters

- **tensor** (Tensor) – Tensor to send.
- **dst** (*int*) – Destination rank.
- **group** (*ProcessGroup, optional*) – The process group to work on
- **tag** (*int, optional*) – Tag to match send with remote recv

torch.distributed.**recv**(*tensor*, *src=None*, *group=<object object>*, *tag=0*)
Receives a tensor synchronously.

Parameters

- **tensor** (Tensor) – Tensor to fill with received data.
- **src** (*int, optional*) – Source rank. Will receive from any process if unspecified.
- **group** (*ProcessGroup, optional*) – The process group to work on
- **tag** (*int, optional*) – Tag to match recv with remote send

Returns Sender rank -1, if not part of the group

isend() and *irecv()* return distributed request objects when used. In general, the type of this object is unspecified as they should never be created manually, but they are guaranteed to support two methods:

- `is_completed()` - returns True if the operation has finished
- `wait()` - will block the process until the operation is finished. `is_completed()` is guaranteed to return True once it returns.

torch.distributed.**isend**(*tensor*, *dst*, *group=<object object>*, *tag=0*)
Sends a tensor asynchronously.

Parameters

- **tensor** (Tensor) – Tensor to send.
- **dst** (*int*) – Destination rank.
- **group** (*ProcessGroup, optional*) – The process group to work on

- **tag** (*int, optional*) – Tag to match send with remote recv

Returns A distributed request object. None, if not part of the group

torch.distributed.**irecv**(*tensor*, *src*, *group=<object object>*, *tag=0*)
Receives a tensor asynchronously.

Parameters

- **tensor** (Tensor) – Tensor to fill with received data.
- **src** (*int*) – Source rank.
- **group** (*ProcessGroup, optional*) – The process group to work on
- **tag** (*int, optional*) – Tag to match recv with remote send

Returns A distributed request object. None, if not part of the group

22.6 Synchronous and asynchronous collective operations

Every collective operation function supports the following two kinds of operations:

synchronous operation - the default mode, when `async_op` is set to False. when the function returns, it is guaranteed that the collective operation is performed (not necessarily completed if its a CUDA op since all CUDA ops are asynchronous), and any further function calls depending on the data of the collective operation can be called. In the synchronous mode, the collective function does not return anything

asynchronous operation - when `async_op` is set to True. The collective operation function returns a distributed request object. In general, you dont need to create it manually and it is guaranteed to support two methods:

- `is_completed()` - returns True if the operation has finished
- `wait()` - will block the process until the operation is finished.

22.7 Collective functions

torch.distributed.**broadcast**(*tensor*, *src*, *group=<object object>*, *async_op=False*)
Broadcasts the tensor to the whole group.

`tensor` must have the same number of elements in all processes participating in the collective.

Parameters

- **tensor** (Tensor) – Data to be sent if `src` is the rank of current process, and tensor to be used to save received data otherwise.
- **src** (*int*) – Source rank.
- **group** (*ProcessGroup, optional*) – The process group to work on
- **async_op** (*bool, optional*) – Whether this op should be an async op

Returns Async work handle, if async_op is set to True. None, if not async_op or if not part of the group

torch.distributed.**all_reduce**(*tensor*, *op=ReduceOp.SUM*, *group=<object object>*, *async_op=False*)
Reduces the tensor data across all machines in such a way that all get the final result.

After the call `tensor` is going to be bitwise identical in all processes.

Parameters

- **tensor** (Tensor) – Input and output of the collective. The function operates in-place.
- **op** (*optional*) – One of the values from `torch.distributed.ReduceOp` enum. Specifies an operation used for element-wise reductions.
- **group** (*ProcessGroup, optional*) – The process group to work on
- **async_op** (*bool, optional*) – Whether this op should be an async op

Returns Async work handle, if async_op is set to True. None, if not async_op or if not part of the group

`torch.distributed.`**reduce**(*tensor*, *dst*, *op=ReduceOp.SUM*, *group=<object object>*, *async_op=False*)

Reduces the tensor data across all machines.

Only the process with rank `dst` is going to receive the final result.

Parameters

- **tensor** (Tensor) – Input and output of the collective. The function operates in-place.
- **dst** (*int*) – Destination rank
- **op** (*optional*) – One of the values from `torch.distributed.ReduceOp` enum. Specifies an operation used for element-wise reductions.
- **group** (*ProcessGroup, optional*) – The process group to work on
- **async_op** (*bool, optional*) – Whether this op should be an async op

Returns Async work handle, if async_op is set to True. None, if not async_op or if not part of the group

`torch.distributed.`**all_gather**(*tensor_list*, *tensor*, *group=<object object>*, *async_op=False*)

Gathers tensors from the whole group in a list.

Parameters

- **tensor_list** (*list[Tensor]*) – Output list. It should contain correctly-sized tensors to be used for output of the collective.
- **tensor** (Tensor) – Tensor to be broadcast from current process.
- **group** (*ProcessGroup, optional*) – The process group to work on
- **async_op** (*bool, optional*) – Whether this op should be an async op

Returns Async work handle, if async_op is set to True. None, if not async_op or if not part of the group

`torch.distributed.`**gather**(*tensor*, *gather_list*, *dst*, *group=<object object>*, *async_op=False*)

Gathers a list of tensors in a single process.

Parameters

- **tensor** (Tensor) – Input tensor.
- **gather_list** (*list[Tensor]*) – List of appropriately-sized tensors to use for received data. Required only in the receiving process.
- **dst** (*int*) – Destination rank. Required in all processes except the one that is receiveing the data.
- **group** (*ProcessGroup, optional*) – The process group to work on

- **async_op** (*bool, optional*) – Whether this op should be an async op

Returns Async work handle, if async_op is set to True. None, if not async_op or if not part of the group

torch.distributed.**scatter**(*tensor*, *scatter_list*, *src*, *group=<object object>*, *async_op=False*)

Scatters a list of tensors to all processes in a group.

Each process will receive exactly one tensor and store its data in the `tensor` argument.

Parameters

- **tensor** (Tensor) – Output tensor.
- **scatter_list** (*list[Tensor]*) – List of tensors to scatter. Required only in the process that is sending the data.
- **src** (*int*) – Source rank. Required in all processes except the one that is sending the data.
- **group** (*ProcessGroup, optional*) – The process group to work on
- **async_op** (*bool, optional*) – Whether this op should be an async op

Returns Async work handle, if async_op is set to True. None, if not async_op or if not part of the group

torch.distributed.**barrier**(*group=<object object>*, *async_op=False*)

Synchronizes all processes.

This collective blocks processes until the whole group enters this function, if async_op is False, or if async work handle is called on wait().

Parameters

- **group** (*ProcessGroup, optional*) – The process group to work on
- **async_op** (*bool, optional*) – Whether this op should be an async op

Returns Async work handle, if async_op is set to True. None, if not async_op or if not part of the group

class torch.distributed.**ReduceOp**

An enum-like class of available reduce operations: `SUM`, `PRODUCT`, `MIN`, and `MAX`.

The values of this class can be accessed as attributes, e.g., `ReduceOp.SUM`. They are used in specifying strategies for reduction collectives, e.g., *reduce()*, *all_reduce_multigpu()*, etc.

Members:

SUM

PRODUCT

MIN

MAX

class torch.distributed.**reduce_op**

Deprecated enum-like class for reduction operations: `SUM`, `PRODUCT`, `MIN`, and `MAX`.

ReduceOp is recommended to use instead.

22.8 Multi-GPU collective functions

If you have more than one GPU on each node, when using the NCCL and Gloo backend, *broadcast_multigpu()* *all_reduce_multigpu()* *reduce_multigpu()* and *all_gather_multigpu()* support distributed collective operations among multiple GPUs within each node. These functions can potentially improve the overall distributed training performance and be easily used by passing a list of tensors. Each Tensor in the passed tensor list needs to be on a separate GPU device of the host where the function is called. Note that the length of the tensor list needs to be identical among all the distributed processes. Also note that currently the multi-GPU collective functions are only supported by the NCCL backend.

For example, if the system we use for distributed training has 2 nodes, each of which has 8 GPUs. On each of the 16 GPUs, there is a tensor that we would like to all-reduce. The following code can serve as a reference:

Code running on Node 0

```
import torch
import torch.distributed as dist

dist.init_process_group(backend="nccl",
                        init_method="file:///distributed_test",
                        world_size=2,
                        rank=0)
tensor_list = []
for dev_idx in range(torch.cuda.device_count()):
    tensor_list.append(torch.FloatTensor([1]).cuda(dev_idx))

dist.all_reduce_multigpu(tensor_list)
```

Code running on Node 1

```
import torch
import torch.distributed as dist

dist.init_process_group(backend="nccl",
                        init_method="file:///distributed_test",
                        world_size=2,
                        rank=1)
tensor_list = []
for dev_idx in range(torch.cuda.device_count()):
    tensor_list.append(torch.FloatTensor([1]).cuda(dev_idx))

dist.all_reduce_multigpu(tensor_list)
```

After the call, all 16 tensors on the two nodes will have the all-reduced value of 16

torch.distributed.**broadcast_multigpu**(*tensor_list*, *src*, *group=<object object>*, *async_op=False*, *src_tensor=0*)

Broadcasts the tensor to the whole group with multiple GPU tensors per node.

`tensor` must have the same number of elements in all the GPUs from all processes participating in the collective. each tensor in the list must be on a different GPU

Only nccl and gloo backend are currently supported tensors should only be GPU tensors

Parameters

- **tensor_list** (*List[Tensor]*) – Tensors that participate in the collective operation. If `src` is the rank, then the specified `src_tensor` element of `tensor_list` (`tensor_list[src_tensor]`) will be broadcast to all other tensors (on different

GPUs) in the src process and all tensors in `tensor_list` of other non-src processes. You also need to make sure that `len(tensor_list)` is the same for all the distributed processes calling this function.

- **src** (*int*) – Source rank.
- **group** (*ProcessGroup, optional*) – The process group to work on
- **async_op** (*bool, optional*) – Whether this op should be an async op
- **src_tensor** (*int, optional*) – Source tensor rank within `tensor_list`

Returns Async work handle, if async_op is set to True. None, if not async_op or if not part of the group

`torch.distributed.`**all_reduce_multigpu**(*tensor_list*, *op=ReduceOp.SUM*, *group=<object object>*, *async_op=False*)

Reduces the tensor data across all machines in such a way that all get the final result. This function reduces a number of tensors on every node, while each tensor resides on different GPUs. Therefore, the input tensor in the tensor list needs to be GPU tensors. Also, each tensor in the tensor list needs to reside on a different GPU.

After the call, all `tensor` in `tensor_list` is going to be bitwise identical in all processes.

Only nccl and gloo backend is currently supported tensors should only be GPU tensors

Parameters

- **list** (*tensor*) – List of input and output tensors of the collective. The function operates in-place and requires that each tensor to be a GPU tensor on different GPUs. You also need to make sure that `len(tensor_list)` is the same for all the distributed processes calling this function.
- **op** (*optional*) – One of the values from `torch.distributed.ReduceOp` enum. Specifies an operation used for element-wise reductions.
- **group** (*ProcessGroup, optional*) – The process group to work on
- **async_op** (*bool, optional*) – Whether this op should be an async op

Returns Async work handle, if async_op is set to True. None, if not async_op or if not part of the group

`torch.distributed.`**reduce_multigpu**(*tensor_list*, *dst*, *op=ReduceOp.SUM*, *group=<object object>*, *async_op=False*, *dst_tensor=0*)

Reduces the tensor data on multiple GPUs across all machines. Each tensor in `tensor_list` should reside on a separate GPU

Only the GPU of `tensor_list[dst_tensor]` on the process with rank `dst` is going to receive the final result.

Only nccl backend is currently supported tensors should only be GPU tensors

Parameters

- **tensor_list** (*List[Tensor]*) – Input and output GPU tensors of the collective. The function operates in-place. You also need to make sure that `len(tensor_list)` is the same for all the distributed processes calling this function.
- **dst** (*int*) – Destination rank
- **op** (*optional*) – One of the values from `torch.distributed.ReduceOp` enum. Specifies an operation used for element-wise reductions.
- **group** (*ProcessGroup, optional*) – The process group to work on
- **async_op** (*bool, optional*) – Whether this op should be an async op

- **dst_tensor** (*int, optional*) – Destination tensor rank within `tensor_list`

Returns Async work handle, if async_op is set to True. None, otherwise

`torch.distributed.`**all_gather_multigpu**(*output_tensor_lists*, *input_tensor_list*, *group=<object object>*, *async_op=False*)

Gathers tensors from the whole group in a list. Each tensor in `tensor_list` should reside on a separate GPU

Only nccl backend is currently supported tensors should only be GPU tensors

Parameters

- **output_tensor_lists** (*List[List[Tensor]]*) – Output lists. It should contain correctly-sized tensors on each GPU to be used for output of the collective. e.g. `output_tensor_lists[i]` contains the all_gather result that resides on the GPU of `input_tensor_list[i]`. Note that each element of `output_tensor_lists[i]` has the size of `world_size * len(input_tensor_list)`, since the function all gathers the result from every single GPU in the group. To interpret each element of `output_tensor_list[i]`, note that `input_tensor_list[j]` of rank k will be appear in `output_tensor_list[i][rank * world_size + j]` Also note that `len(output_tensor_lists)`, and the size of each element in `output_tensor_lists` (each element is a list, therefore `len(output_tensor_lists[i])`) need to be the same for all the distributed processes calling this function.
- **input_tensor_list** (*List[Tensor]*) – List of tensors(on different GPUs) to be broadcast from current process. Note that `len(input_tensor_list)` needs to be the same for all the distributed processes calling this function.
- **group** (*ProcessGroup, optional*) – The process group to work on
- **async_op** (*bool, optional*) – Whether this op should be an async op

Returns Async work handle, if async_op is set to True. None, if not async_op or if not part of the group

22.9 Launch utility

The *torch.distributed* package also provides a launch utility in *torch.distributed.launch*. This helper utility can be used to launch multiple processes per node for distributed training. This utility also supports both python2 and python3.

torch.distributed.launch is a module that spawns up multiple distributed training processes on each of the training nodes.

The utility can be used for single-node distributed training, in which one or more processes per node will be spawned. The utility can be used for either CPU training or GPU training. If the utility is used for GPU training, each distributed process will be operating on a single GPU. This can achieve well-improved single-node training performance. It can also be used in multi-node distributed training, by spawning up multiple processes on each node for well-improved multi-node distributed training performance as well. This will especially be benefitial for systems with multiple Infiniband interfaces that have direct-GPU support, since all of them can be utilized for aggregated communication bandwidth.

In both cases of single-node distributed training or multi-node distributed training, this utility will launch the given number of processes per node (`--nproc_per_node`). If used for GPU training, this number needs to be less or euqal to the number of GPUs on the current system (`nproc_per_node`), and each process will be operating on a single GPU from *GPU 0 to GPU (nproc_per_node - 1)*.

How to use this module:

1. Single-Node multi-process distributed training

```
>>> python -m torch.distributed.launch --nproc_per_node=NUM_GPUS_YOU_HAVE
           YOUR_TRAINING_SCRIPT.py (--arg1 --arg2 --arg3 and all other
           arguments of your training script)
```

2. Multi-Node multi-process distributed training: (e.g. two nodes)

Node 1: *(IP: 192.168.1.1, and has a free port: 1234)*

```
>>> python -m torch.distributed.launch --nproc_per_node=NUM_GPUS_YOU_HAVE
           --nnodes=2 --node_rank=0 --master_addr="192.168.1.1"
           --master_port=1234 YOUR_TRAINING_SCRIPT.py (--arg1 --arg2 --arg3
           and all other arguments of your training script)
```

Node 2:

```
>>> python -m torch.distributed.launch --nproc_per_node=NUM_GPUS_YOU_HAVE
           --nnodes=2 --node_rank=1 --master_addr="192.168.1.1"
           --master_port=1234 YOUR_TRAINING_SCRIPT.py (--arg1 --arg2 --arg3
           and all other arguments of your training script)
```

3. To look up what optional arguments this module offers:

```
>>> python -m torch.distributed.launch --help
```

Important Notices:

1. This utilty and multi-process distributed (single-node or multi-node) GPU training currently only achieves the best performance using the NCCL distributed backend. Thus NCCL backend is the recommended backend to use for GPU training.

2. In your training program, you must parse the command-line argument: `--local_rank=LOCAL_PROCESS_RANK`, which will be provided by this module. If your training program uses GPUs, you should ensure that your code only runs on the GPU device of LOCAL_PROCESS_RANK. This can be done by:

Parsing the local_rank argument

```
>>> import argparse
>>> parser = argparse.ArgumentParser()
>>> parser.add_argument("--local_rank", type=int)
>>> args = parser.parse_args()
```

Set your device to local rank using either

```
>>> torch.cuda.set_device(arg.local_rank)  # before your code runs
```

or

```
>>> with torch.cuda.device(arg.local_rank):
>>>     # your code to run
```

3. In your training program, you are supposed to call the following function at the beginning to start the distributed backend. You need to make sure that the init_method uses `env://`, which is the only supported `init_method` by this module.

```
torch.distributed.init_process_group(backend='YOUR BACKEND',
                                     init_method='env://')
```

4. In your training program, you can either use regular distributed functions or use *torch.nn.parallel.DistributedDataParallel()* module. If your training program uses GPUs for training and you would like to use *torch.nn.parallel.DistributedDataParallel()* module, here is how to configure it.

```
model = torch.nn.parallel.DistributedDataParallel(model,
                                                   device_ids=[arg.local_rank],
                                                   output_device=arg.local_rank)
```

Please ensure that `device_ids` argument is set to be the only GPU device id that your code will be operating on. This is generally the local rank of the process. In other words, the `device_ids` needs to be `[args.local_rank]`, and `output_device` needs to be `args.local_rank` in order to use this utility

5. Another way to pass `local_rank` to the subprocesses via environment variable `LOCAL_RANK`. This behavior is enabled when you launch the script with `--use_env=True`. You must adjust the subprocess example above to replace `args.local_rank` with `os.environ['LOCAL_RANK']`; the launcher will not pass `--local_rank` when you specify this flag.

Warning: `local_rank` is NOT globally unique: it is only unique per process on a machine. Thus, dont use it to decide if you should, e.g., write to a networked filesystem. See https://github.com/pytorch/pytorch/issues/12042 for an example of how things can go wrong if you dont do this correctly.

22.10 Spawn utility

The multiprocessing-doc package also provides a `spawn` function in `torch.multiprocessing.spawn()`. This helper function can be used to spawn multiple processes. It works by passing in the function that you want to run and spawns N processes to run it. This can be used for multiprocess distributed training as well.

For references on how to use it, please refer to PyTorch example - ImageNet implementation

Note that this function requires Python 3.4 or higher.

CHAPTER

TWENTYTHREE

PROBABILITY DISTRIBUTIONS - TORCH.DISTRIBUTIONS

The `distributions` package contains parameterizable probability distributions and sampling functions. This allows the construction of stochastic computation graphs and stochastic gradient estimators for optimization. This package generally follows the design of the TensorFlow Distributions package.

It is not possible to directly backpropagate through random samples. However, there are two main methods for creating surrogate functions that can be backpropagated through. These are the score function estimator/likelihood ratio estimator/REINFORCE and the pathwise derivative estimator. REINFORCE is commonly seen as the basis for policy gradient methods in reinforcement learning, and the pathwise derivative estimator is commonly seen in the reparameterization trick in variational autoencoders. Whilst the score function only requires the value of samples $f(x)$, the pathwise derivative requires the derivative $f'(x)$. The next sections discuss these two in a reinforcement learning example. For more details see Gradient Estimation Using Stochastic Computation Graphs .

23.1 Score function

When the probability density function is differentiable with respect to its parameters, we only need `sample()` and `log_prob()` to implement REINFORCE:

$$\Delta\theta = \alpha r \frac{\partial \log p(a|\pi^\theta(s))}{\partial \theta}$$

where θ are the parameters, α is the learning rate, r is the reward and $p(a|\pi^\theta(s))$ is the probability of taking action a in state s given policy π^θ.

In practice we would sample an action from the output of a network, apply this action in an environment, and then use `log_prob` to construct an equivalent loss function. Note that we use a negative because optimizers use gradient descent, whilst the rule above assumes gradient ascent. With a categorical policy, the code for implementing REINFORCE would be as follows:

```
probs = policy_network(state)
# Note that this is equivalent to what used to be called multinomial
m = Categorical(probs)
action = m.sample()
next_state, reward = env.step(action)
loss = -m.log_prob(action) * reward
loss.backward()
```

23.2 Pathwise derivative

The other way to implement these stochastic/policy gradients would be to use the reparameterization trick from the `rsample()` method, where the parameterized random variable can be constructed via a parameterized deterministic

function of a parameter-free random variable. The reparameterized sample therefore becomes differentiable. The code for implementing the pathwise derivative would be as follows:

```
params = policy_network(state)
m = Normal(*params)
# Any distribution with .has_rsample == True could work based on the application
action = m.rsample()
next_state, reward = env.step(action)  # Assuming that reward is differentiable
loss = -reward
loss.backward()
```

23.3 Distribution

class torch.distributions.distribution.**Distribution**(*batch_shape=torch.Size([])*, *event_shape=torch.Size([])*, *validate_args=None*)

Bases: object

Distribution is the abstract base class for probability distributions.

arg_constraints
Returns a dictionary from argument names to *Constraint* objects that should be satisfied by each argument of this distribution. Args that are not tensors need not appear in this dict.

batch_shape
Returns the shape over which parameters are batched.

cdf(*value*)
Returns the cumulative density/mass function evaluated at *value*.

Parameters **value** (Tensor) –

entropy()
Returns entropy of distribution, batched over batch_shape.

Returns Tensor of shape batch_shape.

enumerate_support(*expand=True*)
Returns tensor containing all values supported by a discrete distribution. The result will enumerate over dimension 0, so the shape of the result will be *(cardinality,) + batch_shape + event_shape* (where *event_shape = ()* for univariate distributions).

Note that this enumerates over all batched tensors in lock-step *[[0, 0], [1, 1],]*. With *expand=False*, enumeration happens along dim 0, but with the remaining batch dimensions being singleton dimensions, *[[0], [1], ...*

To iterate over the full Cartesian product use *itertools.product(m.enumerate_support())*.

Parameters **expand** (*bool*) – whether to expand the support over the batch dims to match the distributions *batch_shape*.

Returns Tensor iterating over dimension 0.

event_shape
Returns the shape of a single sample (without batching).

expand(*batch_shape*, *_instance=None*)
Returns a new distribution instance (or populates an existing instance provided by a derived class) with batch dimensions expanded to *batch_shape*. This method calls *expand* on the distributions parameters.

As such, this does not allocate new memory for the expanded distribution instance. Additionally, this does not repeat any args checking or parameter broadcasting in *__init__.py*, when an instance is first created.

Parameters

- **batch_shape** (*torch.Size*) – the desired expanded size.
- **_instance** – new instance provided by subclasses that need to override *.expand*.

Returns New distribution instance with batch dimensions expanded to *batch_size*.

icdf(*value*)
Returns the inverse cumulative density/mass function evaluated at *value*.

Parameters **value** (Tensor) –

log_prob(*value*)
Returns the log of the probability density/mass function evaluated at *value*.

Parameters **value** (Tensor) –

mean
Returns the mean of the distribution.

perplexity()
Returns perplexity of distribution, batched over batch_shape.

Returns Tensor of shape batch_shape.

rsample(*sample_shape=torch.Size([])*)
Generates a sample_shape shaped reparameterized sample or sample_shape shaped batch of reparameterized samples if the distribution parameters are batched.

sample(*sample_shape=torch.Size([])*)
Generates a sample_shape shaped sample or sample_shape shaped batch of samples if the distribution parameters are batched.

sample_n(*n*)
Generates n samples or n batches of samples if the distribution parameters are batched.

stddev
Returns the standard deviation of the distribution.

support
Returns a *Constraint* object representing this distributions support.

variance
Returns the variance of the distribution.

23.4 ExponentialFamily

class torch.distributions.exp_family.**ExponentialFamily**(*batch_shape=torch.Size([])*, *event_shape=torch.Size([])*, *validate_args=None*)
Bases: *torch.distributions.distribution.Distribution*

ExponentialFamily is the abstract base class for probability distributions belonging to an exponential family, whose probability mass/density function has the form is defined below

$$p_F(x;\theta) = \exp(\langle t(x), \theta\rangle - F(\theta) + k(x))$$

where θ denotes the natural parameters, $t(x)$ denotes the sufficient statistic, $F(\theta)$ is the log normalizer function for a given family and $k(x)$ is the carrier measure.

Note: This class is an intermediary between the *Distribution* class and distributions which belong to an exponential family mainly to check the correctness of the *.entropy()* and analytic KL divergence methods. We use this class to compute the entropy and KL divergence using the AD framework and Bregman divergences (courtesy of: Frank Nielsen and Richard Nock, Entropies and Cross-entropies of Exponential Families).

---

entropy()
Method to compute the entropy using Bregman divergence of the log normalizer.

23.5 Bernoulli

class torch.distributions.bernoulli.**Bernoulli**(*probs=None*, *logits=None*, *validate_args=None*)
Bases: *torch.distributions.exp_family.ExponentialFamily*

Creates a Bernoulli distribution parameterized by *probs* or *logits* (but not both).

Samples are binary (0 or 1). They take the value *1* with probability *p* and *0* with probability *1 - p*.

Example:

```
>>> m = Bernoulli(torch.tensor([0.3]))
>>> m.sample()  # 30% chance 1; 70% chance 0
tensor([ 0.])
```

Parameters

- **probs** (*Number, Tensor*) – the probability of sampling *1*
- **logits** (*Number, Tensor*) – the log-odds of sampling *1*

arg_constraints = {'logits': Real(), 'probs': Interval(lower_bound=0.0, upper_bound=

entropy()

enumerate_support(*expand=True*)

expand(*batch_shape*, *_instance=None*)

has_enumerate_support = True

log_prob(*value*)

logits

mean

param_shape

probs

sample(*sample_shape=torch.Size([])*)

support = Boolean()

variance

23.6 Beta

class torch.distributions.beta.**Beta**(*concentration1*, *concentration0*, *validate_args=None*)

Bases: *torch.distributions.exp_family.ExponentialFamily*

Beta distribution parameterized by *concentration1* and *concentration0*.

Example:

```
>>> m = Beta(torch.tensor([0.5]), torch.tensor([0.5]))
>>> m.sample()  # Beta distributed with concentration concentration1 and␣
↪concentration0
tensor([ 0.1046])
```

Parameters

- **concentration1** (*float or Tensor*) – 1st concentration parameter of the distribution (often referred to as alpha)
- **concentration0** (*float or Tensor*) – 2nd concentration parameter of the distribution (often referred to as beta)

arg_constraints = {'concentration0': GreaterThan(lower_bound=0.0), 'concentration1':

concentration0

concentration1

entropy()

expand(*batch_shape*, *_instance=None*)

has_rsample = True

log_prob(*value*)

mean

rsample(*sample_shape=()*)

support = Interval(lower_bound=0.0, upper_bound=1.0)

variance

23.7 Binomial

class torch.distributions.binomial.**Binomial**(*total_count=1*, *probs=None*, *logits=None*, *validate_args=None*)

Bases: *torch.distributions.distribution.Distribution*

Creates a Binomial distribution parameterized by `total_count` and either *probs* or *logits* (but not both). `total_count` must be broadcastable with *probs*/*logits*.

Example:

```
>>> m = Binomial(100, torch.tensor([0 , .2, .8, 1]))
>>> x = m.sample()
tensor([   0.,   22.,   71.,  100.])

>>> m = Binomial(torch.tensor([[5.], [10.]]), torch.tensor([0.5, 0.8]))
```

(continues on next page)

(continued from previous page)

```
>>> x = m.sample()
tensor([[ 4.,  5.],
        [ 7.,  6.]])
```

Parameters

- **total_count** (*int or Tensor*) – number of Bernoulli trials
- **probs** (Tensor) – Event probabilities
- **logits** (Tensor) – Event log-odds

arg_constraints = {'logits': Real(), 'probs': Interval(lower_bound=0.0, upper_bound=

enumerate_support(*expand=True*)

expand(*batch_shape*, *_instance=None*)

has_enumerate_support = True

log_prob(*value*)

logits

mean

param_shape

probs

sample(*sample_shape=torch.Size([])*)

support

variance

23.8 Categorical

class torch.distributions.categorical.**Categorical**(*probs=None*, *logits=None*, *validate_args=None*)

Bases: *torch.distributions.distribution.Distribution*

Creates a categorical distribution parameterized by either *probs* or *logits* (but not both).

Note: It is equivalent to the distribution that *torch.multinomial()* samples from.

Samples are integers from $\{0, \ldots, K-1\}$ where K is `probs.size(-1)`.

If *probs* is 1D with length-K, each element is the relative probability of sampling the class at that index.

If *probs* is 2D, it is treated as a batch of relative probability vectors.

Note: *probs* must be non-negative, finite and have a non-zero sum, and it will be normalized to sum to 1.

See also: *torch.multinomial()*

Example:

```
>>> m = Categorical(torch.tensor([ 0.25, 0.25, 0.25, 0.25 ]))
>>> m.sample()  # equal probability of 0, 1, 2, 3
tensor(3)
```

Parameters

- **probs** (Tensor) – event probabilities
- **logits** (Tensor) – event log probabilities

arg_constraints = {'logits': Real(), 'probs': Simplex()}

entropy()

enumerate_support(*expand=True*)

expand(*batch_shape*, *_instance=None*)

has_enumerate_support = True

log_prob(*value*)

logits

mean

param_shape

probs

sample(*sample_shape=torch.Size([])*)

support

variance

23.9 Cauchy

class torch.distributions.cauchy.**Cauchy**(*loc*, *scale*, *validate_args=None*)

Bases: *torch.distributions.distribution.Distribution*

Samples from a Cauchy (Lorentz) distribution. The distribution of the ratio of independent normally distributed random variables with means *0* follows a Cauchy distribution.

Example:

```
>>> m = Cauchy(torch.tensor([0.0]), torch.tensor([1.0]))
>>> m.sample()  # sample from a Cauchy distribution with loc=0 and scale=1
tensor([ 2.3214])
```

Parameters

- **loc** (*float or* Tensor) – mode or median of the distribution.
- **scale** (*float or* Tensor) – half width at half maximum.

arg_constraints = {'loc': Real(), 'scale': GreaterThan(lower_bound=0.0)}

cdf(*value*)

entropy()

expand(*batch_shape*, *_instance=None*)

has_rsample = True

icdf(*value*)

log_prob(*value*)

mean

rsample(*sample_shape=torch.Size([])*)

support = Real()

variance

23.10 Chi2

class torch.distributions.chi2.**Chi2**(*df*, *validate_args=None*)

Bases: *torch.distributions.gamma.Gamma*

Creates a Chi2 distribution parameterized by shape parameter *df*. This is exactly equivalent to `Gamma(alpha=0.5*df, beta=0.5)`

Example:

```
>>> m = Chi2(torch.tensor([1.0]))
>>> m.sample()  # Chi2 distributed with shape df=1
tensor([ 0.1046])
```

Parameters **df** (*float or Tensor*) – shape parameter of the distribution

arg_constraints = {'df': GreaterThan(lower_bound=0.0)}

df

expand(*batch_shape*, *_instance=None*)

23.11 Dirichlet

class torch.distributions.dirichlet.**Dirichlet**(*concentration*, *validate_args=None*)

Bases: *torch.distributions.exp_family.ExponentialFamily*

Creates a Dirichlet distribution parameterized by concentration `concentration`.

Example:

```
>>> m = Dirichlet(torch.tensor([0.5, 0.5]))
>>> m.sample()  # Dirichlet distributed with concentrarion concentration
tensor([ 0.1046,  0.8954])
```

Parameters **concentration** (Tensor) – concentration parameter of the distribution (often referred to as alpha)

arg_constraints = {'concentration': GreaterThan(lower_bound=0.0)}

entropy()

expand(*batch_shape*, *_instance=None*)

has_rsample = True

log_prob(*value*)

mean

rsample(*sample_shape=()*)

support = Simplex()

variance

23.12 Exponential

class torch.distributions.exponential.**Exponential**(*rate*, *validate_args=None*)

Bases: *torch.distributions.exp_family.ExponentialFamily*

Creates a Exponential distribution parameterized by `rate`.

Example:

```
>>> m = Exponential(torch.tensor([1.0]))
>>> m.sample()  # Exponential distributed with rate=1
tensor([ 0.1046])
```

Parameters **rate** (*float or Tensor*) – rate = 1 / scale of the distribution

arg_constraints = {'rate': GreaterThan(lower_bound=0.0)}

cdf(*value*)

entropy()

expand(*batch_shape*, *_instance=None*)

has_rsample = True

icdf(*value*)

log_prob(*value*)

mean

rsample(*sample_shape=torch.Size([])*)

stddev

support = GreaterThan(lower_bound=0.0)

variance

23.13 FisherSnedecor

class torch.distributions.fishersnedecor.**FisherSnedecor**(*df1*, *df2*, *validate_args=None*)

Bases: *torch.distributions.distribution.Distribution*

Creates a Fisher-Snedecor distribution parameterized by `df1` and `df2`.

Example:

```
>>> m = FisherSnedecor(torch.tensor([1.0]), torch.tensor([2.0]))
>>> m.sample()  # Fisher-Snedecor-distributed with df1=1 and df2=2
tensor([ 0.2453])
```

Parameters

- **df1** (*float or Tensor*) – degrees of freedom parameter 1
- **df2** (*float or Tensor*) – degrees of freedom parameter 2

arg_constraints = {'df1': GreaterThan(lower_bound=0.0), 'df2': GreaterThan(lower_bou

expand(*batch_shape*, *_instance=None*)

has_rsample = True

log_prob(*value*)

mean

rsample(*sample_shape=torch.Size([])*)

support = GreaterThan(lower_bound=0.0)

variance

23.14 Gamma

class torch.distributions.gamma.**Gamma**(*concentration*, *rate*, *validate_args=None*)

Bases: *torch.distributions.exp_family.ExponentialFamily*

Creates a Gamma distribution parameterized by shape `concentration` and `rate`.

Example:

```
>>> m = Gamma(torch.tensor([1.0]), torch.tensor([1.0]))
>>> m.sample()  # Gamma distributed with concentration=1 and rate=1
tensor([ 0.1046])
```

Parameters

- **concentration** (*float or Tensor*) – shape parameter of the distribution (often referred to as alpha)
- **rate** (*float or Tensor*) – rate = 1 / scale of the distribution (often referred to as beta)

arg_constraints = {'concentration': GreaterThan(lower_bound=0.0), 'rate': GreaterTha

entropy()

expand(*batch_shape*, *_instance=None*)

has_rsample = True

log_prob(*value*)

mean

rsample(*sample_shape=torch.Size([])*)

support = GreaterThan(lower_bound=0.0)

variance

23.15 Geometric

class torch.distributions.geometric.**Geometric**(*probs=None*, *logits=None*, *validate_args=None*)

Bases: *torch.distributions.distribution.Distribution*

Creates a Geometric distribution parameterized by *probs*, where *probs* is the probability of success of Bernoulli trials. It represents the probability that in $k + 1$ Bernoulli trials, the first k trials failed, before seeing a success.

Samples are non-negative integers [0, inf).

Example:

```
>>> m = Geometric(torch.tensor([0.3]))
>>> m.sample()  # underlying Bernoulli has 30% chance 1; 70% chance 0
tensor([ 2.])
```

Parameters

- **probs** (*Number, Tensor*) – the probability of sampling *1*. Must be in range (0, 1]
- **logits** (*Number, Tensor*) – the log-odds of sampling *1*.

arg_constraints = {'logits': Real(), 'probs': Interval(lower_bound=0.0, upper_bound=

entropy()

expand(*batch_shape*, *_instance=None*)

log_prob(*value*)

logits

mean

probs

sample(*sample_shape=torch.Size([])*)

support = IntegerGreaterThan(lower_bound=0)

variance

23.16 Gumbel

class torch.distributions.gumbel.**Gumbel**(*loc*, *scale*, *validate_args=None*)

Bases: *torch.distributions.transformed_distribution.TransformedDistribution*

Samples from a Gumbel Distribution.

Examples:

```
>>> m = Gumbel(torch.tensor([1.0]), torch.tensor([2.0]))
>>> m.sample()  # sample from Gumbel distribution with loc=1, scale=2
tensor([ 1.0124])
```

Parameters

- **loc** (*float or Tensor*) – Location parameter of the distribution
- **scale** (*float or Tensor*) – Scale parameter of the distribution

arg_constraints = {'loc': Real(), 'scale': GreaterThan(lower_bound=0.0)}

entropy()

expand(*batch_shape*, *_instance=None*)

log_prob(*value*)

mean

stddev

support = Real()

variance

23.17 HalfCauchy

class torch.distributions.half_cauchy.**HalfCauchy**(*scale*, *validate_args=None*)

Bases: *torch.distributions.transformed_distribution.TransformedDistribution*

Creates a half-normal distribution parameterized by *scale* where:

```
X ~ Cauchy(0, scale)
Y = |X| ~ HalfCauchy(scale)
```

Example:

```
>>> m = HalfCauchy(torch.tensor([1.0]))
>>> m.sample()  # half-cauchy distributed with scale=1
tensor([ 2.3214])
```

Parameters **scale** (*float or Tensor*) – scale of the full Cauchy distribution

arg_constraints = {'scale': GreaterThan(lower_bound=0.0)}

cdf(*value*)

entropy()

expand(*batch_shape*, *_instance=None*)

has_rsample = True

icdf(*prob*)

log_prob(*value*)

mean

scale

support = GreaterThan(lower_bound=0.0)

variance

23.18 HalfNormal

class torch.distributions.half_normal.**HalfNormal**(*scale*, *validate_args=None*)

Bases: *torch.distributions.transformed_distribution.TransformedDistribution*

Creates a half-normal distribution parameterized by *scale* where:

```
X ~ Normal(0, scale)
Y = |X| ~ HalfNormal(scale)
```

Example:

```
>>> m = HalfNormal(torch.tensor([1.0]))
>>> m.sample()  # half-normal distributed with scale=1
tensor([ 0.1046])
```

Parameters **scale** (*float or Tensor*) – scale of the full Normal distribution

arg_constraints = {'scale': GreaterThan(lower_bound=0.0)}

cdf(*value*)

entropy()

expand(*batch_shape*, *_instance=None*)

has_rsample = True

icdf(*prob*)

log_prob(*value*)

mean

scale

support = GreaterThan(lower_bound=0.0)

variance

23.19 Independent

class torch.distributions.independent.**Independent**(*base_distribution*, *reinterpreted_batch_ndims*, *validate_args=None*)

Bases: *torch.distributions.distribution.Distribution*

Reinterprets some of the batch dims of a distribution as event dims.

This is mainly useful for changing the shape of the result of *log_prob()*. For example to create a diagonal Normal distribution with the same shape as a Multivariate Normal distribution (so they are interchangeable), you can:

```
>>> loc = torch.zeros(3)
>>> scale = torch.ones(3)
>>> mvn = MultivariateNormal(loc, scale_tril=torch.diag(scale))
>>> [mvn.batch_shape, mvn.event_shape]
[torch.Size(()), torch.Size((3,))]
>>> normal = Normal(loc, scale)
>>> [normal.batch_shape, normal.event_shape]
[torch.Size((3,)), torch.Size(())]
>>> diagn = Independent(normal, 1)
>>> [diagn.batch_shape, diagn.event_shape]
[torch.Size(()), torch.Size((3,))]
```

Parameters

- **base_distribution** (torch.distributions.distribution.Distribution) – a base distribution
- **reinterpreted_batch_ndims** (*int*) – the number of batch dims to reinterpret as event dims

arg_constraints = {}

entropy()

enumerate_support(*expand=True*)

expand(*batch_shape*, *_instance=None*)

has_enumerate_support

has_rsample

log_prob(*value*)

mean

rsample(*sample_shape=torch.Size([])*)

sample(*sample_shape=torch.Size([])*)

support

variance

23.20 Laplace

class torch.distributions.laplace.**Laplace**(*loc*, *scale*, *validate_args=None*)

Bases: *torch.distributions.distribution.Distribution*

Creates a Laplace distribution parameterized by loc and :attr:scale.

Example:

```
>>> m = Laplace(torch.tensor([0.0]), torch.tensor([1.0]))
>>> m.sample()  # Laplace distributed with loc=0, scale=1
tensor([ 0.1046])
```

Parameters

- **loc** (*float or Tensor*) – mean of the distribution

- **scale** (*float or Tensor*) – scale of the distribution

arg_constraints = {'loc': Real(), 'scale': GreaterThan(lower_bound=0.0)}

cdf(*value*)

entropy()

expand(*batch_shape*, *_instance=None*)

has_rsample = True

icdf(*value*)

log_prob(*value*)

mean

rsample(*sample_shape=torch.Size([])*)

stddev

support = Real()

variance

23.21 LogNormal

class torch.distributions.log_normal.**LogNormal**(*loc*, *scale*, *validate_args=None*)

Bases: *torch.distributions.transformed_distribution.TransformedDistribution*

Creates a log-normal distribution parameterized by *loc* and *scale* where:

```
X ~ Normal(loc, scale)
Y = exp(X) ~ LogNormal(loc, scale)
```

Example:

```
>>> m = LogNormal(torch.tensor([0.0]), torch.tensor([1.0]))
>>> m.sample()  # log-normal distributed with mean=0 and stddev=1
tensor([ 0.1046])
```

Parameters

- **loc** (*float or Tensor*) – mean of log of distribution
- **scale** (*float or Tensor*) – standard deviation of log of the distribution

arg_constraints = {'loc': Real(), 'scale': GreaterThan(lower_bound=0.0)}

entropy()

expand(*batch_shape*, *_instance=None*)

has_rsample = True

loc

mean

scale

support = GreaterThan(lower_bound=0.0)

variance

23.22 LowRankMultivariateNormal

class torch.distributions.lowrank_multivariate_normal.**LowRankMultivariateNormal**(*loc*, *cov_factor*, *cov_diag*, *validate_args=None*)

Bases: *torch.distributions.distribution.Distribution*

Creates a multivariate normal distribution with covariance matrix having a low-rank form parameterized by `cov_factor` and `cov_diag`:

```
covariance_matrix = cov_factor @ cov_factor.T + cov_diag
```

Example

```
>>> m = LowRankMultivariateNormal(torch.zeros(2), torch.tensor([1, 0]), torch.
↪tensor([1, 1]))
>>> m.sample()  # normally distributed with mean=`[0,0]`, cov_factor=`[1,0]`, cov_
↪diag=`[1,1]`
tensor([-0.2102, -0.5429])
```

Parameters

- **loc** (Tensor) – mean of the distribution with shape *batch_shape* + *event_shape*
- **cov_factor** (Tensor) – factor part of low-rank form of covariance matrix with shape *batch_shape* + *event_shape* + *(rank,)*
- **cov_diag** (Tensor) – diagonal part of low-rank form of covariance matrix with shape *batch_shape* + *event_shape*

Note: The computation for determinant and inverse of covariance matrix is avoided when *cov_factor.shape[1] << cov_factor.shape[0]* thanks to Woodbury matrix identity and matrix determinant lemma. Thanks to these formulas, we just need to compute the determinant and inverse of the small size capacitance matrix:

```
capacitance = I + cov_factor.T @ inv(cov_diag) @ cov_factor
```

---

arg_constraints = {'cov_diag': GreaterThan(lower_bound=0.0), 'cov_factor': Real(), '

covariance_matrix

entropy()

expand(*batch_shape*, *_instance=None*)

has_rsample = True

log_prob(*value*)

mean

precision_matrix

rsample(*sample_shape=torch.Size([])*)

scale_tril

support = Real()

variance

23.23 Multinomial

class torch.distributions.multinomial.**Multinomial**(*total_count=1*, *probs=None*, *logits=None*, *validate_args=None*)

Bases: *torch.distributions.distribution.Distribution*

Creates a Multinomial distribution parameterized by `total_count` and either *probs* or *logits* (but not both). The innermost dimension of *probs* indexes over categories. All other dimensions index over batches.

Note that `total_count` need not be specified if only *log_prob()* is called (see example below)

Note: *probs* must be non-negative, finite and have a non-zero sum, and it will be normalized to sum to 1.

- *sample()* requires a single shared *total_count* for all parameters and samples.
- *log_prob()* allows different *total_count* for each parameter and sample.

Example:

```
>>> m = Multinomial(100, torch.tensor([ 1., 1., 1., 1.]))
>>> x = m.sample()  # equal probability of 0, 1, 2, 3
tensor([ 21.,  24.,  30.,  25.])

>>> Multinomial(probs=torch.tensor([1., 1., 1., 1.])).log_prob(x)
tensor([-4.1338])
```

Parameters

- **total_count** (*int*) – number of trials
- **probs** (Tensor) – event probabilities
- **logits** (Tensor) – event log probabilities

arg_constraints = {'logits': Real(), 'probs': Simplex()}

expand(*batch_shape*, *_instance=None*)

log_prob(*value*)

logits

mean

param_shape

probs

sample(*sample_shape=torch.Size([])*)

support

variance

23.24 MultivariateNormal

class torch.distributions.multivariate_normal.**MultivariateNormal**(*loc*, *covariance_matrix=None*, *precision_matrix=None*, *scale_tril=None*, *validate_args=None*)

Bases: *torch.distributions.distribution.Distribution*

Creates a multivariate normal (also called Gaussian) distribution parameterized by a mean vector and a covariance matrix.

The multivariate normal distribution can be parameterized either in terms of a positive definite covariance matrix $\mathbf{\Sigma}$ or a positive definite precision matrix $\mathbf{\Sigma}^{-1}$ or a lower-triangular matrix $\mathbf{L}$ with positive-valued diagonal entries, such that $\mathbf{\Sigma} = \mathbf{L}\mathbf{L}^\top$. This triangular matrix can be obtained via e.g. Cholesky decomposition of the covariance.

Example

```
>>> m = MultivariateNormal(torch.zeros(2), torch.eye(2))
>>> m.sample()  # normally distributed with mean=`[0,0]` and covariance_matrix=`I`
tensor([-0.2102, -0.5429])
```

Parameters

- **loc** (Tensor) – mean of the distribution
- **covariance_matrix** (Tensor) – positive-definite covariance matrix
- **precision_matrix** (Tensor) – positive-definite precision matrix
- **scale_tril** (Tensor) – lower-triangular factor of covariance, with positive-valued diagonal

---

Note: Only one of *covariance_matrix* or *precision_matrix* or *scale_tril* can be specified.

Using *scale_tril* will be more efficient: all computations internally are based on *scale_tril*. If *covariance_matrix* or *precision_matrix* is passed instead, it is only used to compute the corresponding lower triangular matrices using a Cholesky decomposition.

---

arg_constraints = {'covariance_matrix': PositiveDefinite(), 'loc': RealVector(), 'pr

covariance_matrix

entropy()

expand(*batch_shape*, *_instance=None*)

has_rsample = True

log_prob(*value*)

mean

precision_matrix

rsample(*sample_shape=torch.Size([])*)

scale_tril

support = Real()

variance

23.25 NegativeBinomial

class torch.distributions.negative_binomial.**NegativeBinomial**(*total_count*, *probs=None*, *logits=None*, *validate_args=None*)

Bases: *torch.distributions.distribution.Distribution*

Creates a Negative Binomial distribution, i.e. distribution of the number of independent identical Bernoulli trials needed before `total_count` failures are achieved. The probability of success of each Bernoulli trial is *probs*.

Parameters

- **total_count** (*float or Tensor*) – non-negative number of negative Bernoulli trials to stop, although the distribution is still valid for real valued count
- **probs** (Tensor) – Event probabilities of success in the half open interval [0, 1)
- **logits** (Tensor) – Event log-odds for probabilities of success

arg_constraints = {'logits': Real(), 'probs': HalfOpenInterval(lower_bound=0.0, upper

expand(*batch_shape*, *_instance=None*)

log_prob(*value*)

logits

mean

param_shape

probs

sample(*sample_shape=torch.Size([])*)

support = IntegerGreaterThan(lower_bound=0)

variance

23.26 Normal

class torch.distributions.normal.**Normal**(*loc*, *scale*, *validate_args=None*)

Bases: *torch.distributions.exp_family.ExponentialFamily*

Creates a normal (also called Gaussian) distribution parameterized by `loc` and `scale`.

Example:

```
>>> m = Normal(torch.tensor([0.0]), torch.tensor([1.0]))
>>> m.sample()  # normally distributed with loc=0 and scale=1
tensor([ 0.1046])
```

Parameters

- **loc** (*float or Tensor*) – mean of the distribution (often referred to as mu)
- **scale** (*float or Tensor*) – standard deviation of the distribution (often referred to as sigma)

arg_constraints = {'loc': Real(), 'scale': GreaterThan(lower_bound=0.0)}

cdf(*value*)

entropy()

expand(*batch_shape*, *_instance=None*)

has_rsample = True

icdf(*value*)

log_prob(*value*)

mean

rsample(*sample_shape=torch.Size([])*)

sample(*sample_shape=torch.Size([])*)

stddev

support = Real()

variance

23.27 OneHotCategorical

class torch.distributions.one_hot_categorical.**OneHotCategorical**(*probs=None*, *logits=None*, *validate_args=None*)

Bases: *torch.distributions.distribution.Distribution*

Creates a one-hot categorical distribution parameterized by *probs* or *logits*.

Samples are one-hot coded vectors of size `probs.size(-1)`.

Note: *probs* must be non-negative, finite and have a non-zero sum, and it will be normalized to sum to 1.

See also: `torch.distributions.Categorical()` for specifications of *probs* and *logits*.

Example:

```
>>> m = OneHotCategorical(torch.tensor([ 0.25, 0.25, 0.25, 0.25 ]))
>>> m.sample()  # equal probability of 0, 1, 2, 3
tensor([ 0.,  0.,  0.,  1.])
```

Parameters

- **probs** (Tensor) – event probabilities
- **logits** (Tensor) – event log probabilities

arg_constraints = {'logits': Real(), 'probs': Simplex()}

entropy()

enumerate_support(*expand=True*)

expand(*batch_shape*, *_instance=None*)

has_enumerate_support = True

log_prob(*value*)

logits

mean

param_shape

probs

sample(*sample_shape=torch.Size([])*)

support = Simplex()

variance

23.28 Pareto

class torch.distributions.pareto.**Pareto**(*scale*, *alpha*, *validate_args=None*)

Bases: *torch.distributions.transformed_distribution.TransformedDistribution*

Samples from a Pareto Type 1 distribution.

Example:

```
>>> m = Pareto(torch.tensor([1.0]), torch.tensor([1.0]))
>>> m.sample()  # sample from a Pareto distribution with scale=1 and alpha=1
tensor([ 1.5623])
```

Parameters

- **scale** (*float or* Tensor) – Scale parameter of the distribution
- **alpha** (*float or* Tensor) – Shape parameter of the distribution

arg_constraints = {'alpha': GreaterThan(lower_bound=0.0), 'scale': GreaterThan(lower

entropy()

expand(*batch_shape*, *_instance=None*)

mean

support

variance

23.29 Poisson

class torch.distributions.poisson.**Poisson**(*rate*, *validate_args=None*)

Bases: *torch.distributions.exp_family.ExponentialFamily*

Creates a Poisson distribution parameterized by `rate`, the rate parameter.

Samples are nonnegative integers, with a pmf given by

$$\text{rate}^k \frac{e^{-\text{rate}}}{k!}$$

Example:

```
>>> m = Poisson(torch.tensor([4]))
>>> m.sample()
tensor([ 3.])
```

Parameters **rate** (*Number, Tensor*) – the rate parameter

arg_constraints = {'rate': GreaterThan(lower_bound=0.0)}

expand(*batch_shape*, *_instance=None*)

log_prob(*value*)

mean

sample(*sample_shape=torch.Size([])*)

support = IntegerGreaterThan(lower_bound=0)

variance

23.30 RelaxedBernoulli

class torch.distributions.relaxed_bernoulli.**RelaxedBernoulli**(*temperature*, *probs=None*, *logits=None*, *validate_args=None*)

Bases: *torch.distributions.transformed_distribution.TransformedDistribution*

Creates a RelaxedBernoulli distribution, parametrized by *temperature*, and either *probs* or *logits* (but not both). This is a relaxed version of the *Bernoulli* distribution, so the values are in (0, 1), and has reparametrizable samples.

Example:

```
>>> m = RelaxedBernoulli(torch.tensor([2.2]),
                         torch.tensor([0.1, 0.2, 0.3, 0.99]))
>>> m.sample()
tensor([ 0.2951,  0.3442,  0.8918,  0.9021])
```

Parameters

- **temperature** (Tensor) – relaxation temperature
- **probs** (*Number, Tensor*) – the probability of sampling *1*

- **logits** (*Number, Tensor*) – the log-odds of sampling *1*

arg_constraints = {'logits': Real(), 'probs': Interval(lower_bound=0.0, upper_bound=

expand(*batch_shape*, *_instance=None*)

has_rsample = True

logits

probs

support = Interval(lower_bound=0.0, upper_bound=1.0)

temperature

23.31 LogitRelaxedBernoulli

class torch.distributions.relaxed_bernoulli.**LogitRelaxedBernoulli**(*temperature*, *probs=None*, *logits=None*, *validate_args=None*)

Bases: *torch.distributions.distribution.Distribution*

Creates a LogitRelaxedBernoulli distribution parameterized by *probs* or *logits* (but not both), which is the logit of a RelaxedBernoulli distribution.

Samples are logits of values in (0, 1). See [1] for more details.

Parameters

- **temperature** (Tensor) – relaxation temperature
- **probs** (*Number, Tensor*) – the probability of sampling *1*
- **logits** (*Number, Tensor*) – the log-odds of sampling *1*

[1] The Concrete Distribution: A Continuous Relaxation of Discrete Random Variables (Maddison et al, 2017)

[2] Categorical Reparametrization with Gumbel-Softmax (Jang et al, 2017)

arg_constraints = {'logits': Real(), 'probs': Interval(lower_bound=0.0, upper_bound=

expand(*batch_shape*, *_instance=None*)

log_prob(*value*)

logits

param_shape

probs

rsample(*sample_shape=torch.Size([])*)

support = Real()

23.32 RelaxedOneHotCategorical

class torch.distributions.relaxed_categorical.**RelaxedOneHotCategorical**(*temperature*, *probs=None*, *logits=None*, *validate_args=None*)

Bases: *torch.distributions.transformed_distribution.TransformedDistribution*

Creates a RelaxedOneHotCategorical distribution parametrized by *temperature*, and either *probs* or *logits*. This is a relaxed version of the `OneHotCategorical` distribution, so its samples are on simplex, and are reparametrizable.

Example:

```
>>> m = RelaxedOneHotCategorical(torch.tensor([2.2]),
                                 torch.tensor([0.1, 0.2, 0.3, 0.4]))
>>> m.sample()
tensor([ 0.1294,  0.2324,  0.3859,  0.2523])
```

Parameters

- **temperature** (`Tensor`) – relaxation temperature
- **probs** (`Tensor`) – event probabilities
- **logits** (`Tensor`) – the log probability of each event.

arg_constraints = {'logits': Real(), 'probs': Simplex()}

expand(*batch_shape*, *_instance=None*)

has_rsample = True

logits

probs

support = Simplex()

temperature

23.33 StudentT

class torch.distributions.studentT.**StudentT**(*df*, *loc=0.0*, *scale=1.0*, *validate_args=None*)

Bases: *torch.distributions.distribution.Distribution*

Creates a Students t-distribution parameterized by degree of freedom `df`, mean `loc` and scale `scale`.

Example:

```
>>> m = StudentT(torch.tensor([2.0]))
>>> m.sample()  # Student's t-distributed with degrees of freedom=2
tensor([ 0.1046])
```

Parameters

- **df** (*float or Tensor*) – degrees of freedom
- **loc** (*float or Tensor*) – mean of the distribution
- **scale** (*float or Tensor*) – scale of the distribution

arg_constraints = {'df': GreaterThan(lower_bound=0.0), 'loc': Real(), 'scale': Grea

entropy()

expand(*batch_shape*, *_instance=None*)

has_rsample = True

log_prob(*value*)

mean

rsample(*sample_shape=torch.Size([])*)

support = Real()

variance

23.34 TransformedDistribution

class torch.distributions.transformed_distribution.**TransformedDistribution**(*base_distribution*, *transforms*, *validate_args=None*)

Bases: *torch.distributions.distribution.Distribution*

Extension of the Distribution class, which applies a sequence of Transforms to a base distribution. Let f be the composition of transforms applied:

```
X ~ BaseDistribution
Y = f(X) ~ TransformedDistribution(BaseDistribution, f)
log p(Y) = log p(X) + log |det (dX/dY)|
```

Note that the `.event_shape` of a *TransformedDistribution* is the maximum shape of its base distribution and its transforms, since transforms can introduce correlations among events.

An example for the usage of *TransformedDistribution* would be:

```
# Building a Logistic Distribution
# X ~ Uniform(0, 1)
# f = a + b * logit(X)
# Y ~ f(X) ~ Logistic(a, b)
base_distribution = Uniform(0, 1)
transforms = [SigmoidTransform().inv, AffineTransform(loc=a, scale=b)]
logistic = TransformedDistribution(base_distribution, transforms)
```

For more examples, please look at the implementations of *Gumbel*, *HalfCauchy*, *HalfNormal*, *LogNormal*, *Pareto*, *Weibull*, *RelaxedBernoulli* and *RelaxedOneHotCategorical*

arg_constraints = {}

cdf(*value*)
Computes the cumulative distribution function by inverting the transform(s) and computing the score of the base distribution.

expand(*batch_shape*, *_instance=None*)

has_rsample

icdf(*value*)
Computes the inverse cumulative distribution function using transform(s) and computing the score of the base distribution.

log_prob(*value*)
Scores the sample by inverting the transform(s) and computing the score using the score of the base distribution and the log abs det jacobian.

rsample(*sample_shape=torch.Size([])*)
Generates a sample_shape shaped reparameterized sample or sample_shape shaped batch of reparameterized samples if the distribution parameters are batched. Samples first from base distribution and applies *transform()* for every transform in the list.

sample(*sample_shape=torch.Size([])*)
Generates a sample_shape shaped sample or sample_shape shaped batch of samples if the distribution parameters are batched. Samples first from base distribution and applies *transform()* for every transform in the list.

support

23.35 Uniform

class torch.distributions.uniform.**Uniform**(*low*, *high*, *validate_args=None*)
Bases: *torch.distributions.distribution.Distribution*

Generates uniformly distributed random samples from the half-open interval `[low, high)`.

Example:

```
>>> m = Uniform(torch.tensor([0.0]), torch.tensor([5.0]))
>>> m.sample()  # uniformly distributed in the range [0.0, 5.0)
tensor([ 2.3418])
```

Parameters

- **low** (*float or Tensor*) – lower range (inclusive).
- **high** (*float or Tensor*) – upper range (exclusive).

arg_constraints = {'high': Dependent(), 'low': Dependent()}

cdf(*value*)

entropy()

expand(*batch_shape*, *_instance=None*)

has_rsample = True

icdf(*value*)

log_prob(*value*)

mean

rsample(*sample_shape=torch.Size([])*)

stddev

support

variance

23.36 Weibull

class torch.distributions.weibull.**Weibull**(*scale*, *concentration*, *validate_args=None*)
Bases: *torch.distributions.transformed_distribution.TransformedDistribution*

Samples from a two-parameter Weibull distribution.

Example

```
>>> m = Weibull(torch.tensor([1.0]), torch.tensor([1.0]))
>>> m.sample()  # sample from a Weibull distribution with scale=1, concentration=1
tensor([ 0.4784])
```

Parameters

- **scale** (*float or Tensor*) – Scale parameter of distribution (lambda).
- **concentration** (*float or Tensor*) – Concentration parameter of distribution (k/shape).

arg_constraints = {'concentration': GreaterThan(lower_bound=0.0), 'scale': GreaterTha

entropy()

expand(*batch_shape*, *_instance=None*)

mean

support = GreaterThan(lower_bound=0.0)

variance

23.37 KL Divergence

torch.distributions.kl.**kl_divergence**(*p*, *q*)
Compute Kullback-Leibler divergence $KL(p\|q)$ between two distributions.

$$KL(p\|q) = \int p(x) \log \frac{p(x)}{q(x)} \, dx$$

Parameters

- **p** (Distribution) – A Distribution object.
- **q** (Distribution) – A Distribution object.

Returns A batch of KL divergences of shape *batch_shape*.

Return type *Tensor*

Raises `NotImplementedError` – If the distribution types have not been registered via *register_kl()*.

`torch.distributions.kl.`**register_kl**(*type_p*, *type_q*)

Decorator to register a pairwise function with *kl_divergence()*. Usage:

```
@register_kl(Normal, Normal)
def kl_normal_normal(p, q):
    # insert implementation here
```

Lookup returns the most specific (type,type) match ordered by subclass. If the match is ambiguous, a *RuntimeWarning* is raised. For example to resolve the ambiguous situation:

```
@register_kl(BaseP, DerivedQ)
def kl_version1(p, q): ...
@register_kl(DerivedP, BaseQ)
def kl_version2(p, q): ...
```

you should register a third most-specific implementation, e.g.:

```
register_kl(DerivedP, DerivedQ)(kl_version1)  # Break the tie.
```

Parameters

- **type_p** (*type*) – A subclass of `Distribution`.
- **type_q** (*type*) – A subclass of `Distribution`.

23.38 *Transforms*

class `torch.distributions.transforms.`**Transform**(*cache_size=0*)

Abstract class for invertable transformations with computable log det jacobians. They are primarily used in `torch.distributions.TransformedDistribution`.

Caching is useful for tranforms whose inverses are either expensive or numerically unstable. Note that care must be taken with memoized values since the autograd graph may be reversed. For example while the following works with or without caching:

```
y = t(x)
t.log_abs_det_jacobian(x, y).backward()  # x will receive gradients.
```

However the following will error when caching due to dependency reversal:

```
y = t(x)
z = t.inv(y)
grad(z.sum(), [y])  # error because z is x
```

Derived classes should implement one or both of `_call()` or `_inverse()`. Derived classes that set *bijective=True* should also implement *log_abs_det_jacobian()*.

Parameters **cache_size** (*int*) – Size of cache. If zero, no caching is done. If one, the latest single value is cached. Only 0 and 1 are supported.

Variables

- **domain** (*Constraint*) – The constraint representing valid inputs to this transform.

- **codomain** (*Constraint*) – The constraint representing valid outputs to this transform which are inputs to the inverse transform.
- **bijective** (*bool*) – Whether this transform is bijective. A transform `t` is bijective iff `t.inv(t(x)) == x` and `t(t.inv(y)) == y` for every `x` in the domain and `y` in the codomain. Transforms that are not bijective should at least maintain the weaker pseudoinverse properties `t(t.inv(t(x)) == t(x)` and `t.inv(t(t.inv(y))) == t.inv(y)`.
- **sign** (*int or Tensor*) – For bijective univariate transforms, this should be +1 or -1 depending on whether transform is monotone increasing or decreasing.
- **event_dim** (*int*) – Number of dimensions that are correlated together in the transform `event_shape`. This should be 0 for pointwise transforms, 1 for transforms that act jointly on vectors, 2 for transforms that act jointly on matrices, etc.

inv
Returns the inverse *Transform* of this transform. This should satisfy `t.inv.inv is t`.

sign
Returns the sign of the determinant of the Jacobian, if applicable. In general this only makes sense for bijective transforms.

log_abs_det_jacobian(*x*, *y*)
Computes the log det jacobian *log |dy/dx|* given input and output.

class torch.distributions.transforms.**ComposeTransform**(*parts*)
Composes multiple transforms in a chain. The transforms being composed are responsible for caching.

Parameters **parts** (list of *Transform*) – A list of transforms to compose.

class torch.distributions.transforms.**ExpTransform**(*cache_size=0*)
Transform via the mapping $y = \exp(x)$.

class torch.distributions.transforms.**PowerTransform**(*exponent*, *cache_size=0*)
Transform via the mapping $y = x^{\text{exponent}}$.

class torch.distributions.transforms.**SigmoidTransform**(*cache_size=0*)
Transform via the mapping $y = \frac{1}{1+\exp(-x)}$ and $x = \text{logit}(y)$.

class torch.distributions.transforms.**AbsTransform**(*cache_size=0*)
Transform via the mapping $y = |x|$.

class torch.distributions.transforms.**AffineTransform**(*loc*, *scale*, *event_dim=0*, *cache_size=0*)
Transform via the pointwise affine mapping $y = \text{loc} + \text{scale} \times x$.

Parameters

- **loc** (*Tensor or float*) – Location parameter.
- **scale** (*Tensor or float*) – Scale parameter.
- **event_dim** (*int*) – Optional size of *event_shape*. This should be zero for univariate random variables, 1 for distributions over vectors, 2 for distributions over matrices, etc.

class torch.distributions.transforms.**SoftmaxTransform**(*cache_size=0*)
Transform from unconstrained space to the simplex via $y = \exp(x)$ then normalizing.

This is not bijective and cannot be used for HMC. However this acts mostly coordinate-wise (except for the final normalization), and thus is appropriate for coordinate-wise optimization algorithms.

class torch.distributions.transforms.**StickBreakingTransform**(*cache_size=0*)
Transform from unconstrained space to the simplex of one additional dimension via a stick-breaking process.

This transform arises as an iterated sigmoid transform in a stick-breaking construction of the *Dirichlet* distribution: the first logit is transformed via sigmoid to the first probability and the probability of everything else, and then the process recurses.

This is bijective and appropriate for use in HMC; however it mixes coordinates together and is less appropriate for optimization.

class torch.distributions.transforms.**LowerCholeskyTransform**(*cache_size=0*)
Transform from unconstrained matrices to lower-triangular matrices with nonnegative diagonal entries.

This is useful for parameterizing positive definite matrices in terms of their Cholesky factorization.

23.39 *Constraints*

The following constraints are implemented:

- constraints.boolean
- constraints.dependent
- constraints.greater_than(lower_bound)
- constraints.integer_interval(lower_bound, upper_bound)
- constraints.interval(lower_bound, upper_bound)
- constraints.lower_cholesky
- constraints.lower_triangular
- constraints.nonnegative_integer
- constraints.positive
- constraints.positive_definite
- constraints.positive_integer
- constraints.real
- constraints.real_vector
- constraints.simplex
- constraints.unit_interval

class torch.distributions.constraints.**Constraint**
Abstract base class for constraints.

A constraint object represents a region over which a variable is valid, e.g. within which a variable can be optimized.

check(*value*)
Returns a byte tensor of *sample_shape* + *batch_shape* indicating whether each event in value satisfies this constraint.

torch.distributions.constraints.**dependent_property**
alias of torch.distributions.constraints._DependentProperty

torch.distributions.constraints.**integer_interval**
alias of torch.distributions.constraints._IntegerInterval

torch.distributions.constraints.**greater_than**
alias of torch.distributions.constraints._GreaterThan

torch.distributions.constraints.**greater_than_eq**
alias of torch.distributions.constraints._GreaterThanEq

torch.distributions.constraints.**less_than**
alias of torch.distributions.constraints._LessThan

torch.distributions.constraints.**interval**
alias of torch.distributions.constraints._Interval

torch.distributions.constraints.**half_open_interval**
alias of torch.distributions.constraints._HalfOpenInterval

23.40 *Constraint Registry*

PyTorch provides two global *ConstraintRegistry* objects that link *Constraint* objects to *Transform* objects. These objects both input constraints and return transforms, but they have different guarantees on bijectivity.

1. `biject_to(constraint)` looks up a bijective *Transform* from `constraints.real` to the given `constraint`. The returned transform is guaranteed to have `.bijective = True` and should implement `.log_abs_det_jacobian()`.
2. `transform_to(constraint)` looks up a not-necessarily bijective *Transform* from `constraints.real` to the given `constraint`. The returned transform is not guaranteed to implement `.log_abs_det_jacobian()`.

The `transform_to()` registry is useful for performing unconstrained optimization on constrained parameters of probability distributions, which are indicated by each distributions `.arg_constraints` dict. These transforms often overparameterize a space in order to avoid rotation; they are thus more suitable for coordinate-wise optimization algorithms like Adam:

```
loc = torch.zeros(100, requires_grad=True)
unconstrained = torch.zeros(100, requires_grad=True)
scale = transform_to(Normal.arg_constraints['scale'])(unconstrained)
loss = -Normal(loc, scale).log_prob(data).sum()
```

The `biject_to()` registry is useful for Hamiltonian Monte Carlo, where samples from a probability distribution with constrained `.support` are propagated in an unconstrained space, and algorithms are typically rotation invariant.:

```
dist = Exponential(rate)
unconstrained = torch.zeros(100, requires_grad=True)
sample = biject_to(dist.support)(unconstrained)
potential_energy = -dist.log_prob(sample).sum()
```

Note: An example where `transform_to` and `biject_to` differ is `constraints.simplex`: `transform_to(constraints.simplex)` returns a *SoftmaxTransform* that simply exponentiates and normalizes its inputs; this is a cheap and mostly coordinate-wise operation appropriate for algorithms like SVI. In contrast, `biject_to(constraints.simplex)` returns a *StickBreakingTransform* that bijects its input down to a one-fewer-dimensional space; this a more expensive less numerically stable transform but is needed for algorithms like HMC.

The `biject_to` and `transform_to` objects can be extended by user-defined constraints and transforms using their `.register()` method either as a function on singleton constraints:

```
transform_to.register(my_constraint, my_transform)
```

or as a decorator on parameterized constraints:

```
@transform_to.register(MyConstraintClass)
def my_factory(constraint):
    assert isinstance(constraint, MyConstraintClass)
    return MyTransform(constraint.param1, constraint.param2)
```

You can create your own registry by creating a new *ConstraintRegistry* object.

class torch.distributions.constraint_registry.**ConstraintRegistry**
Registry to link constraints to transforms.

register(*constraint*, *factory=None*)
Registers a *Constraint* subclass in this registry. Usage:

```
@my_registry.register(MyConstraintClass)
def construct_transform(constraint):
    assert isinstance(constraint, MyConstraint)
    return MyTransform(constraint.arg_constraints)
```

Parameters

- **constraint** (subclass of *Constraint*) – A subclass of *Constraint*, or a singleton object of the desired class.
- **factory** (*callable*) – A callable that inputs a constraint object and returns a *Transform* object.

CHAPTER

TWENTYFOUR

TORCH.HUB

Pytorch Hub is a pre-trained model repository designed to facilitate research reproducibility.

24.1 Publishing models

Pytorch Hub supports publishing pre-trained models(model definitions and pre-trained weights) to a github repository by adding a simple `hubconf.py` file;

`hubconf.py` can have multiple entrypoints. Each entrypoint is defined as a python function (example: a pre-trained model you want to publish).

```
def entrypoint_name(*args, **kwargs):
    # args & kwargs are optional, for models which take positional/keyword arguments.
    ...
```

24.1.1 How to implement an entrypoint?

Here is a code snippet specifies an entrypoint for `resnet18` model if we expand the implementation in `pytorch/vision/hubconf.py`. In most case importing the right function in `hubconf.py` is sufficient. Here we just want to use the expanded version as an example to show how it works. You can see the full script in pytorch/vision repo

```
dependencies = ['torch']
from torchvision.models.resnet import resnet18 as _resnet18

# resnet18 is the name of entrypoint
def resnet18(pretrained=False, **kwargs):
    """ # This docstring shows up in hub.help()
    Resnet18 model
    pretrained (bool): kwargs, load pretrained weights into the model
    """
    # Call the model, load pretrained weights
    model = _resnet18(pretrained=pretrained, **kwargs)
    return model
```

- `dependencies` variable is a **list** of package names required to **load** the model. Note this might be slightly different from dependencies required for training a model.
- `args` and `kwargs` are passed along to the real callable function.
- Docstring of the function works as a help message. It explains what does the model do and what are the allowed positional/keyword arguments. Its highly recommended to add a few examples here.

- Entrypoint function can either return a model(nn.module), or auxiliary tools to make the user workflow smoother, e.g. tokenizers.
- Callables prefixed with underscore are considered as helper functions which wont show up in `torch.hub.list()`.
- Pretrained weights can either be stored locally in the github repo, or loadable by `torch.hub.load_state_dict_from_url()`. If less than 2GB, its recommended to attach it to a project release and use the url from the release. In the example above `torchvision.models.resnet.resnet18` handles `pretrained`, alternatively you can put the following logic in the entrypoint definition.

```
if pretrained:
    # For checkpoint saved in local github repo, e.g. <RELATIVE_PATH_TO_CHECKPOINT>
↪=weights/save.pth
    dirname = os.path.dirname(__file__)
    checkpoint = os.path.join(dirname, <RELATIVE_PATH_TO_CHECKPOINT>)
    state_dict = torch.load(checkpoint)
    model.load_state_dict(state_dict)

    # For checkpoint saved elsewhere
    checkpoint = 'https://download.pytorch.org/models/resnet18-5c106cde.pth'
    model.load_state_dict(torch.hub.load_state_dict_from_url(checkpoint,␣
↪progress=False))
```

24.1.2 Important Notice

- The published models should be at least in a branch/tag. It cant be a random commit.

24.2 Loading models from Hub

Pytorch Hub provides convenient APIs to explore all available models in hub through `torch.hub.list()`, show docstring and examples through `torch.hub.help()` and load the pre-trained models using `torch.hub.load()`

`torch.hub.load`(*github*, *model*, *force_reload=False*, **args*, ***kwargs*)

Load a model from a github repo, with pretrained weights.

Parameters

- **github** – Required, a string with format repo_owner/repo_name[:tag_name] with an optional tag/branch. The default branch is *master* if not specified. Example: pytorch/vision[:hub]
- **model** – Required, a string of entrypoint name defined in repos hubconf.py
- **force_reload** – Optional, whether to discard the existing cache and force a fresh download. Default is *False*.
- ***args** – Optional, the corresponding args for callable *model*.
- ****kwargs** – Optional, the corresponding kwargs for callable *model*.

Returns a single model with corresponding pretrained weights.

24.2.1 Running a loaded model:

Note that `*args, **kwargs` in `torch.load()` are used to **instantiate** a model. After you loaded a model, how can you find out what you can do with the model? A suggested workflow is

- `dir(model)` to see all avaialble methods of the model.
- `help(model.foo)` to check what arguments `model.foo` takes to run

To help users explore without refering to documentation back and forth, we strongly recommend repo owners make function help messages clear and succinct. Its also helpful to include a minimal working example.

24.2.2 Where are my downloaded models saved?

The locations are used in the order of

- Calling `hub.set_dir(<PATH_TO_HUB_DIR>)`
- `$TORCH_HOME/hub`, if environment variable `TORCH_HOME` is set.
- `$XDG_CACHE_HOME/torch/hub`, if environment variable `XDG_CACHE_HOME` is set.
- `~/.cache/torch/hub`

`torch.hub.`**`set_dir`**(*d*)

Optionally set hub_dir to a local dir to save downloaded models & weights.

If this argument is not set, env variable *TORCH_HUB_DIR* will be searched first, *~/.torch/hub* will be created and used as fallback.

Parameters **d** – path to a local folder to save downloaded models & weights.

24.2.3 Caching logic

By default, we dont clean up files after loading it. Hub uses the cache by default if it already exists in `hub_dir`.

Users can force a reload by calling `hub.load(..., force_reload=True)`. This will delete the existing github folder and downloaded weights, reinitialize a fresh download. This is useful when updates are published to the same branch, users can keep up with the latest release.

24.2.4 Known limitations:

Torch hub works by importing the package as if it was installed. Therere some side effects introduced by importing in Python. For example, you can see new items in Python caches `sys.modules` and `sys.path_importer_cache` which is normal Python behavior.

A known limitation that worth mentioning here is user **CANNOT** load two different branches of the same repo in the **same python process**. Its just like installing two packages with the same name in Python, which is not good. Cache might join the party and give you surprises if you actually try that. Of course its totally fine to load them in separate processes.