

目錄

PyTorch 0.4 中文文档	1.1
笔记	1.2
自动求导机制	1.2.1
广播语义	1.2.2
CUDA语义	1.2.3
扩展PyTorch	1.2.4
常见问题	1.2.5
多进程最佳实践	1.2.6
序列化语义	1.2.7
Windows 常见问题	1.2.8
包参考	1.3
Torch	1.3.1
torch.Tensor	1.3.2
Tensor Attributes	1.3.3
torch.sparse	1.3.4
torch.cuda	1.3.5
torch.Storage	1.3.6
torch.nn	1.3.7
torch.nn.functional	1.3.8
自动差异化包 - torch.autograd	1.3.9
torch.optim	1.3.10
torch.nn.init	1.3.11
torch.distributions	1.3.12
Multiprocessing包 - torch.multiprocessing	1.3.13
分布式通讯包 - torch.distributed	1.3.14
torch.utils.bottleneck	1.3.15
torch.utils.checkpoint	1.3.16
torch.utils.cpp_extension	1.3.17
torch.utils.data	1.3.18
torch.utils.ffi	1.3.19
torch.utils.model_zoo	1.3.20
torch.onnx	1.3.21
遗留包 - torch.legacy	1.3.22
torchvision 参考	1.4
torchvision	1.4.1

torchvision.datasets	1.4.2
torchvision.models	1.4.3
torchvision.transform	1.4.4
torchvision.utils	1.4.5

PyTorch 0.4 中文文档



维护组织: [PyTorch 中文网](#)

PyTorch 是一个针对深度学习, 并且使用 GPU 和 CPU 来优化的 tensor library (张量库).

- [在线阅读](#)
- [PDF格式](#)
- [EPUB格式](#)
- [MOBI格式](#)
- [代码仓库](#)

笔记

自动求导机制

1. 从后向中排除子图
 - `requires_grad`
 - `volatile`
2. 自动求导如何编码历史信息
3. `Variable`上的In-place操作
4. In-place正确性检查

本说明将概述Autograd如何工作并记录操作。没有必要全部了解，但建议您熟悉它，他可以将帮助你编写程序更高效，更清洁；同时还可以帮助您进行调试。

向后排除子视图：

每个变量都有一个标记：`requires_grad` 允许从梯度计算中细分排除子图，并可以提高效率。

`requires_grad`

如果一个输入变量定义 `requires_grad` ,那么他的输出也可以使用 `requires_grad` ；相反，只有当所有的输入变量都不定义 `requires_grad` 梯度，才不会输出梯度。如果其中所有的变量都不需要计算梯度，在子图中从不执行向后计算。

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

当您想要冻结部分模型时，这个标志特别有用;除非您事先知道不会使用到某些参数的梯度。

例如，如果您想调整预训练的 CNN ，只要切换冻结模型中的 `requires_grad` 标志即可，直到计算到最后一层才会保存中间缓冲区，仿射变换和网络输出都需要使用梯度的权值。

```

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)

```

autograd如何编码历史信息：

每个变量都有一个 `.creator` 属性，它指向把它作为输出的函数。这是一个由 `Function` 对象作为节点组成的有向无环图（DAG）的入口点，它们之间的引用就是图的边。每次执行一个操作时，一个表示它的新 `Function` 就被实例化，它的 `forward()` 方法被调用，并且它输出的 `Variable` 的创建者被设置为这个 `Function`。然后，通过跟踪从任何变量到叶节点的路径，可以重建创建数据的操作序列，并自动计算梯度。

需要注意的一点是，整个图在每次迭代时都是从头开始重新创建的，这就允许使用任意的Python控制流语句，这样可以在每次迭代时改变图的整体形状和大小。在启动训练之前不必对所有可能的路径进行编码——**what you run is what you differentiate**。

Variable上的In-place操作：

支持自动归档中的就地操作是一件很困难的事情，我们在大多数情况下都不鼓励使用它们。**Autograd**的积极缓冲区释放和重用使其非常高效，并且在现场操作实际上会降低内存使用量的情况下，极少数场合很少。除非您在内存压力很大的情况下运行，否则您可能永远不需要使用它们。


限制现场操作适用性的两个主要原因：

1. 覆盖计算梯度所需的值。这就是为什么变量不支持 `log_`。其梯度公式需要原始输入，而通过计算逆运算可以重新创建它，它在数值上是不稳定的，并且需要额外的工作，这往往会失败使用这些功能的目的。
2. 每个 `in-place` 操作实际上需要实现重写计算图。不合适的版本只需分配新对象，并保留对旧图的引用，而 `in-place` 操作则需要将所有输入的 `creator` 更改为 `Function` 表示此操作。这就比较棘手，特别是如果有许多变量引用相同的存储（例如通过索引或转置创建的），并且如果被修改输入的存储被其他 `Variable` 引用，则 `in-place` 函数实际上会抛出错误。

In-place正确性检测：

每个变量都保留一个版本计数器 `version counter`，当在任何操作中被使用时，它都会递增。当函数保存任何用于后向的 `tensor` 时，还会保存其包含变量的版本计数器 `version counter`。一旦访问，`self.saved_tensors` 它被会被检查，如果它大于保存的值，则会引起错误。

译者署名

用户名	头像	职能	签名
Song		翻译	人生总要追求点什么

广播语义

- 一般语义
- 直接语义
- 向后兼容性

许多 `pytorch` 操作都支持 `NumPy` 广播语义

简而言之,如果 `Pytorch` 操作支持广播,则其张量参数可以自动扩展为相同大小 (不需要复制数据)。

一般语义

如果 `pytorch` 张量满足以下条件,那么就可以广播:

- 每个张量至少有一个维度。
- 在遍历维度大小时,从尾部维度开始遍历,并且二者维度必须相等,它们其中一个要么是 1 要么不存在。

例如:

```
>>> x=torch.FloatTensor(5,7,3)
>>> y=torch.FloatTensor(5,7,3)
# 相同形状的质量可以被广播(上述规则总是成立的)

>>> x=torch.FloatTensor()
>>> y=torch.FloatTensor(2,2)
# x和y不能被广播,因为x没有维度

# can line up trailing dimensions
>>> x=torch.FloatTensor(5,3,4,1)
>>> y=torch.FloatTensor( 3,1,1)
# x和y可以被广播
# 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

# 但是:
>>> x=torch.FloatTensor(5,2,4,1)
>>> y=torch.FloatTensor( 3,1,1)
# x和y不能被广播,因为在`3rd`中
# x and y are not broadcastable, because in the 3rd trailing dimension 2 != 3
```

如果 `x` 和 `y` 可以被广播,得到的张量大小的计算方法如下:

- 如果维数 `x` 和 `y` 不相等，在维度较少的张量的维度前加上 `1` 使它们相等的长度。
- 然后,对于每个维度的大小,生成维度的大小是 `x` 和 `y` 的最大值。

例如:

```
# 可以排列尾部维度,使阅读更容易
>>> x=torch.FloatTensor(5,1,4,1)
>>> y=torch.FloatTensor( 3,1,1)
>>> (x+y).size()
torch.Size([5, 3, 4, 1])

# 但不是必要的:
>>> x=torch.FloatTensor(1)
>>> y=torch.FloatTensor(3,1,7)
>>> (x+y).size()
torch.Size([3, 1, 7])

>>> x=torch.FloatTensor(5,2,4,1)
>>> y=torch.FloatTensor(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
```

直接语义(In-place语义)

一个复杂的问题是 in-place 操作不允许 in-place 张量像广播那样改变形状。

例如：

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

# but:
>>> x=torch.FloatTensor(1,3,1)
>>> y=torch.FloatTensor(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.
```

向后兼容性

以前版本的 PyTorch 只要张量中的元素数量是相等的,便允许某些点状 `pointwise` 函数在不同的形状的张量上执行,其中点状操作是通过将每个张量视为 `1` 维来执行。PyTorch 现在支持广播语义并且不推荐使用点状函数操作向

量,并且张量不支持广播但具有相同数量的元素将生成一个 Python 警告。

注意,广播的引入可能会导致向后不兼容,因为两个张量的形状不同,但是可以被广播且具有相同数量的元素。

例如:

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

事先生成一个 `torch.Size([4,1])` 的张量,然后再提供一个 `torch.Size([4,4])` 的张量。为了帮助识别你代码中可能存在的由引入广播语义的向后不兼容情况,您可以设置 `torch.utils.backcompat.broadcast_warning.enabled` 为 `True`,这种情况下会生成一个 python 警告。

例如:

```
>>> 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.
```

译者署名

用户名	头像	职能	签名
Song		翻译	人生总要追求点什么

CUDA语义

- 异步执行
 - CUDA流
- 内存管理
- 最佳实践
 - 设备无关代码
 - 使用固定的内存缓冲区
 - 使用 `nn.DataParallel` 替代 `multiprocessing`

3.1 设备分配

`torch.cuda` 用于设置和运行 `CUDA` 操作。它会跟踪当前选定的GPU，并且您分配的所有CUDA张量将默认在该设备上创建。所选设备可以使用 `torch.cuda.device` 环境管理器进行更改。

一旦分配了张量，您就可以对其执行操作而必在意所选的设备如何，并且结果将总是与张量一起放置在相同的设备上。

默认的情况下不允许进行交叉 GPU 操作，除了 `copy_()` 和其他具有类似复制功能的方法（如 `to()` 和 `cuda()`）之外。除非启用端到端的存储器访问，否则任何尝试将张量分配到不同设备上的操作都会引发错误。

下面可以用一个小例子来展示：

```

cuda = torch.device("cuda") # 默认为CUDA设备
cuda0 = torch.device("cuda:0")
cuda2 = torch.device("cuda:2") # GPU 2

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

# 在GPU 1上分配张量
with torch.cuda.device(1):
    # GPU 1
    a = torch.tensor([1., 2.], device=cuda)

    # 从CPU传递到GPU 1
    # a、b所在的设备都是 device(type='cuda', index=1)
    b = torch.tensor([1., 2.]).cuda()

    # 也可以用Tensor.to来传递张量
    # b2的设备与a同
    b2 = torch.tensor([1., 2.]).to(device=cuda)

    # 即使是在环境中，你也可以指定设备
    d = torch.randn(2, device=cuda2)
    e = torch.randn(2).to(cuda2)

```

3.2 异步执行

默认情况下，GPU 操作是异步的。当你调用一个使用 GPU 的函数时，这些操作会在特定的设备上排队，但不一定在稍后执行。这允许我们并行更多的计算，包括 CPU 或其他 GPU 上的操作。

一般情况下，异步计算的效果对调用者是不可见的，因为（1）每个设备按照它们排队的顺序执行操作，（2）在 CPU 和 GPU 之间或两个 GPU 之间复制数据时，PyTorch 自动执行必要的同步。因此，计算将按每个操作同步执行的方式进行。

您可以通过设置环境变量 `CUDA_LAUNCH_BLOCKING = 1` 来强制进行同步计算。当 GPU 发生错误时，这可能非常方便。（使用异步执行，只有在实际执行操作之后才会报告此类错误，因此堆栈跟踪不会显示请求的位置。）

作为一个例外，`copy_()` 等几个函数允许一个显式的异步参数 `async`，它允许调用者在不必要时绕过同步。另一个例外是 CUDA 流，解释如下：

3.2.1 CUDA 流

CUDA 流是属于特定设备的线性执行序列。您通常不需要明确创建一个，默认情况下，每个设备都使用其自己的“默认”流。

除非显式的使用同步函数（例如 `synchronize()` 或 `wait_stream()` ），否则每个流内的操作都按照它们创建的顺序进行序列化，但是来自不同流的操作可以以任意相对顺序并发执行。例如，下面的代码是不正确的：

```
cuda = torch.device("cuda")
s = torch.cuda.stream() # 在当前流中创建一个新的流
A = torch.empty((100,100), device = cuda).normal_(0.0, 1.0)

with torch.cuda.stream(s):
    # sum()可能在normal_()执行完成前就开始执行
    B = torch.sum(A)
```

当“当前流”是默认流时，如上所述，PyTorch 在数据移动时自动执行必要的同步。但是，使用非默认流时，用户有责任确保正确的同步。

3.3 内存管理

PyTorch 使用缓存内存分配器来加速内存分配。这允许在没有设备同步的情况下快速释放内存。但是，由分配器管理的未使用的内存仍将显示为在 `nvidia-smi` 中使用。您可以使用 `memory_allocated()` 和 `max_memory_allocated()` 来监视张量占用的内存，并使用 `memory_cached()` 和 `max_memory_cached()` 来监视由缓存分配器管理的内存。调用 `empty_cache()` 可以从 PyTorch 中释放所有未使用的缓存内存，以便其他 GPU 应用程序可以使用这些内存。但是，被张量占用的 GPU 内存不会被释放，因此它不能为 PyTorch 增加可用的 GPU 内存量。

3.4 最佳实践

3.4.1 设备诊断

由于 PyTorch 的结构，您可能需要明确编写与设备无关的（CPU 或 GPU）代码；比如创建一个新的张量作为循环神经网络的初始隐藏状态。

第一步是确定是否应该使用 GPU。一种常见的模式是使用 Python 的 `argparse` 模块来读入用户参数，并且有一个标志可用于禁用 CUDA，并结合 `is_available()` 使用。在下面的内容中，`args.device` 会生成一个 `torch.device` 对象，该对象可用于将张量移动到 CPU 或 CUDA。

```
import argparse
import torch

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

现在我们有 `args.device`，我们可以使用它在所需的设备上创建一个张量。

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

这可以在许多情况下用于生成设备不可知代码。以下是使用 `dataloader` 的例子：

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

在系统上使用多个 GPU 时，您可以使用 `CUDA_VISIBLE_DEVICES` 环境标志来管理 PyTorch 可用的 GPU。如上所述，要手动控制在哪个 GPU 上创建张量，最佳做法是使用 `torch.cuda.device` 上下文管理器。

```
print("外部的设备是0") # 在设备0上
with torch.cuda.device(1):
    print("内部的设备是1") # 设备1
print("外部的设备仍是0") # 设备0
```

如果您有一个张量，并且想要在同一个设备上创建一个相同类型的张量，那么您可以使用 `torch.Tensor.new_*` 方法（请参阅 `torch.Tensor`）。`torch.Tensor.new_*` 方法保留了设备和张量的其他属性，而前面提到的 `torch.*` 工厂函数（Creation Ops）创建的张量则取决于当前的 GPU 上下文和所传递的属性参数。

这是建立模块时推荐的做法，在前向传递期间需要在内部创建新的张量

```

cuda = torch.device("cuda")
x_cpu = torch.empty(2)
y_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]])
print(y_cpu_long)
      tensor([[ 1,  2,  3]])

```

如果要创建与另一个张量相同类型和大小的张量，并将其填充为1或0，则可以使用 `ones_like()` 或 `zeros_like()` 作为便捷的辅助函数（也可以保留 `torch.device` 和 `torch.dtype` 的张量）。

```

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)

```

3.4 使用固定的内存缓冲区

当数据源自固定（页面锁定）内存时，主机在GPU上的数据副本运行速度会更快。`CPU Tensors` 和存储器暴露了一个 `pin_memory()` 方法，该方法返回对象的一个副本，并将数据放入固定区域。

一旦您固定(pin)一个张量或存储器，您就可以使用异步 GPU 副本。只需将一个额外的 `non_blocking = True` 参数传递给 `cuda()` 调用即可。这可以用于计算与数据传输的并行。

通过将 `pin_memory = True` 传递给其构造函数，可以将 `DataLoader` 返回的批量数据置于固定内存(pin memory)中。

3.5 使用`nn.DataParallel`代替多线程处理

大多数涉及批量输入和多个GPU的使用案例应默认使用 `DataParallel` 来利用多个 GPU。即使使用 GIL，单个 Python 进程也可以使多个 GPU 饱和。

从版本 0.1.9 开始，大量的 GPU（8+）可能未被充分利用。但是，这是一个正在积极开发中的已知问题。像往常一样，测试你的用例。

使用 CUDA 模型进行多线程处理(multiprocessing)存在重要的注意事项；除非准确的满足了数据处理的要求，否则很可能您的程序将具有不正确或未定义的行为。

译者署名

用户名	头像	职能	签名
风中劲草		翻译	人生总要追求点什么

扩展PyTorch

- 扩展 `torch.autograd`
- 扩展 `torch.nn`
 - 增加 `Module`
- 编写自定义C扩展

本篇文章中包含如何扩展 `torch.nn`，`torch.autograd` 和使用 `C` 库来编写自定义的 `C` 扩展工具。

扩展`torch.autograd`

添加操作 `autograd` 需要 `Function` 为每个操作实现一个新的子类。回想一下，`Function` 使用 `autograd` 来计算结果和梯度，并对操作历史进行编码。每个新功能都需要您实现两种方法：

- `forward()` - 执行操作的代码。如果您指定了默认值，则可以根据需求使用任意参数，其中一些参数可选。这里支持各种 `Python` 对象。`Variable` 参数在调用之前会被转换 `Tensor`，并且它们的使用情况将在 `graph` 中注册。请注意，此逻辑不会遍历 `lists / dicts /`和其他任何数据的结构，并且只考虑被直接调用的 `Variables` 参数。如果有多个输出你可以返回单个 `Tensor` 或 `Tensor` 格式的元组。另外，请参阅 `Function` 文档查找只能被 `forward()` 调用的有用方法的说明。
- `backward()` - 计算梯度的公式。它将被赋予与输出一样多的 `Variable` 参数，其中的每一个表示对应梯度的输出。它应该返回与输入一样多的 `Variable`，其中的每一个表示都包含其相应输入的梯度。如果输入不需要计算梯度 (请参阅 `needs_input_grad` 属性)，或者是非 `Variable` 对象，则可返回 `None` 类。此外，如果你在 `forward()` 方法中则可以返回比输入更多的梯度，只要它们都是 `None` 类型即可。

你可以从下面的代码看到 `torch.nn` 模块的 `Linear` 函数，以及注解

```

# Inherit from Function
class Linear(Function):

    # bias is an optional argument
    def forward(self, input, weight, bias=None):
        self.save_for_backward(input, weight, bias)
        output = input.mm(weight.t())
        if bias is not None:
            output += bias.unsqueeze(0).expand_as(output)
        return output

    # This function has only a single output, so it gets only one gradient
    def backward(self, grad_output):
        # This is a pattern that is very convenient - at the top of backward
        # unpack saved_tensors and initialize all gradients w.r.t. inputs to
        # None. Thanks to the fact that additional trailing None's are
        # ignored, the return statement is simple even when the function has
        # optional inputs.
        input, weight, bias = self.saved_tensors
        grad_input = grad_weight = grad_bias = None

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

        return grad_input, grad_weight, grad_bias

```

现在，为了方便使用这些自定义操作，推荐使用 `apply` 方法：

```
linear = LinearFunction.apply
```

我们下面给出一个由非变量参数进行参数化的函数的例子：

```

class MulConstant(Function):
    @staticmethod
    def forward(ctx, tensor, constant):
        # ctx is a context object that can be used to stash info
        # 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

```

你可能想检测你刚刚实现的 `backward` 方法是否正确的计算了梯度。你可以使用小的有限差分法(`Finite Difference`)进行数值估计。

```

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 = (Variable(torch.randn(20,20).double(), requires_grad=True),
         Variable(torch.randn(30,20).double(), requires_grad=True),)
test = gradcheck(Linear.apply, input, eps=1e-6, atol=1e-4)
print(test)

```

扩展 `torch.nn`

`nn` 模块包含两种接口 - `modules` 和他们的功能版本。你可以用两种方法扩展它，但是我们建议，在扩展 `layer` 的时候使用 `modules`，因为 `modules` 保存着参数和 `buffer`。如果使用无参数操作的话，那么建议使用激活函数，池化等函数。

添加操作的功能版本已经在上面的章节中已经介绍了。

增加一个 `Module`。

由于 `nn` 大量使用 `autograd`。所以，添加一个新的 `Module` 类需要实现一个 `Function` 类，它会执行对应的操作并且计算梯度。我们只需要很少的代码就可以实现上面 `Linear` 模块的功能。现在，我们需要实现两个函数：

- `__init__` (optional) - 接收 `kernel sizes` 内核大小，特征数量等参数，并初始化 `parameters` 参数和 `buffers` 缓冲区。
- `forward()` - 实例化 `Function` 并使用它来执行操作。它与上面显示的 `functional wrapper` 非常相似。

下面是实现 `Linear` 模块的方式：

```

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 Variable, that will
        # automatically be registered as Module's parameter once it
        # is 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
            # optional ones can be None if you want.
            self.register_parameter('bias', None)

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

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

```

编写自定义的C扩展

即将发布。不过现在你可以在[GitHub](#)上找到一些例子。

译者署名

用户名	头像	职能	签名
Song		翻译	人生总要追求点什么

常见问题

我的模型报告 "cuda runtime error(2): out of memory"

如错误消息所示，您的 GPU 上的内存不足。由于我们经常在 PyTorch 中处理大量数据，因此小错误可能会迅速导致程序耗尽所有 GPU；幸运的是，这些情况下的修复通常很简单。这里有几个常见的东西要检查：

不要在训练循环中累积历史记录。默认情况下，当计算涉及到有需要梯度的变量时，此计算过程将保留运算的历史记录。这意味着您应该避免在计算中使用这些变量，这些变量的生存期将超出您的训练循环（例如在跟踪统计数据时）。您应该分离该变量或访问其底层数据。

有时，当可微分变量可能发生时，它可能并不明显。考虑以下训练循环（从[源代码](#)节选）：

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

在本例中，由于 `loss` 是具有 `autograd` 历史记录的可微分变量，所以 `total_loss` 将在整个训练循环中累积历史记录。你可以替换成 `total_loss += float(loss)` 来解决这个问题。

这个问题的另一个例子：[1](#)

删除你不需要的张量和变量。如果将一个张量或变量分配到一个局部栈，在局部栈超出作用域之前，Python 都不会将其释放。您可以使用 `del x` 释放该引用。同样，如果将一个张量或变量赋值给对象的成员变量，直到该对象超出作用域之前，它将不会释放。如果你及时删除你不需要的临时变量，你将获得最佳的内存使用率。

作用域的范围可能比你想象的要大。例如：

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

在本段代码中，即使当 `h` 在执行时，`intermediate` 仍然存在，因为它的作用域延伸出了循环的末尾。为了尽早释放它，当你不需要它时，你应该用

```
del intermediate
```

 删除这个中间值。

不要在太大的序列上运行 **RNN**。因为 RNN 反向传播所需的内存量与 RNN 的长度成线性关系；因此，如果尝试向 RNN 提供一个太长的序列时，会耗尽内存。

这一现象的技术术语是时间反向传播 [Backpropagation through time](#)，关于如何实现截断的 BPTT 有很多参考资料，包括在单词语言模型 [word language model](#) 中；截断由 [这个论坛帖子](#) 中描述的 `repackage` 函数处理。

不要使用太大的线性图层。线性层 `nn.Linear(m, n)` 使用 $O(nm)$ 存储器：也就是说，权值的存储器需求随着要素的数量按比例缩放。这种方式会轻易的占用你的内存（并且记住，你将至少需要两倍存储权值的内存量，因为你还需要存储梯度。）

我的 GPU 内存没有正确释放

PyTorch 使用缓存内存分配器来加速内存分配。因此，`nvidia-smi` 中显示的值通常不会反映真实的内存使用情况。有关 GPU 内存管理的更多细节，请参阅 [内存管理](#)。

如果您的 GPU 内存在 Python 退出后仍未释放，那么很可能某些 Python 子进程仍然存在。你可以通过 `ps -elf | grep python` 找到它们，并用

```
kill -9 [pid]
```

 手动杀死它们。

我的多个数据加载器返回相同的随机数

您可能正使用其他库生成数据集中的随机数。例如，当通过 `fork` 启动工作子进程时，NumPy 的 RNG 会被复制。请参阅 [torch.utils.data.DataLoader](#) 的文档，了解如何使用其 `worker_init_fn` 选项正确设置工作进程中的随机种子。

我的回归网络不能使用数据并行

在具有 `DataParallel` 或 `data_parallel()` 的模块中使用

```
pack sequence > recurrent network > unpack sequence
```

 模式时有一个非常微妙的地方。每个设备上的 `forward()` 的输入只会是整个输入的一部分。由于默认情况下，解包操作

```
torch.nn.utils.rnn.pad_packed_sequence()
```

 仅填充到其所见的最长输入，即该特定设备上的最长输入，所以在将结果收集在一起时会发生尺寸的不匹配。因此，您可以利用 `pad_packed_sequence()` 的 `total_length` 参数来确保 `forward()` 调用返回相同长度

的序列。例如，你可以写：


```

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

class MyModule(nn.Module):
    # ... __init__, 以及其他访求

    # padding_input 的形状是[B x T x *] (batch_first 模式)，包含按
    # 长度排序的序列
    # B 是批量大小
    # T 是最大序列长度
    def forward(self, padded_input, input_lengths):
        total_length = padded_input.size(1) # 获取最大序列长度
        packed_input = pack_padded_sequence(padded_input, input_lengths,
                                           batch_first=True)
        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)

```

此外，在批量的维度为dim 1（第1轴）（即 batch_first = False）时需要额外注意数据的并行性。在这种情况下，pack_padded_sequence 函数的第一个参数 padding_input 维度将是 [T x B x *]，并且应该沿dim 1（第1轴）分散，但第二个参数 input_lengths 的维度为 [B]，应该沿dim 0（第0轴）分散。需要额外的代码来操纵张量的维度。

译者署名

用户名	头像	职能	签名
风中劲草		翻译	人生总要追求点什么

多进程最佳实践

- [CUDA 张量的共享](#)
- [最佳实践和技巧](#)
 - [避免和防止死锁](#)
 - [重用通过队列发送的缓冲区](#)
 - [异步多进程训练](#)
 - [hogwild](#)

`torch.multiprocessing` 是 Python 的 `multiprocessing` 多进程模块的替代品。它支持完全相同的操作，但对其进行了扩展，以便所有通过多进程队列 `multiprocessing.Queue` 发送的张量都能将其数据移入共享内存，而且仅将其句柄发送到另一个进程。

注意：

当张量 `Tensor` 被发送到另一个进程时，张量的数据和梯度 `torch.Tensor.grad` 都将被共享。

这一特性允许实现各种训练方法，如 Hogwild，A3C 或任何其他需要异步操作的训练方法。

一、CUDA 张量的共享

仅 Python 3 支持进程之间共享 CUDA 张量，我们可以使用 `spawn` 或 `forkserver` 启动此类方法。Python 2 中的 `multiprocessing` 多进程处理只能使用 `fork` 创建子进程，并且 CUDA 运行时不支持多进程处理。

警告：

CUDA API 规定输出到其他进程的共享张量，只要它们被这些进程使用时，都将持续保持有效。您应该小心并确保您共享的 CUDA 张量不会超出它应该的作用范围（不会出现作用范围延伸的问题）。这对于共享模型的参数应该不是问题，但应该小心地传递其他类型的数据。请注意，此限制不适用于共享的 CPU 内存。

也可以参阅：， [使用 `nn.DataParallel` 替代多进程处理](#)

二、最佳实践和技巧

1、避免和防止死锁

产生新进程时会出现很多错误，导致死锁最常见的原因是后台线程。如果有任何持有锁或导入模块的线程，并且 `fork` 被调用，则子进程很可能处于崩溃状态，并且会以不同方式死锁或失败。请注意，即使您没有这样做，Python 中内置的库也可能，更不必说 `multiprocessing` 了。 `multiprocessing.Queue` 多进程队列实际上

是一个非常复杂的类，它产生了多个用于序列化、发送和接收对象的线程，并且它们也可能导致上述问题。如果您发现自己处于这种情况，请尝试使用 `multiprocessing.queues.SimpleQueue`，它不使用任何其他额外的线程。

我们正在尽可能的为您提供便利，并确保这些死锁不会发生，但有些事情不受我们控制。如果您有任何问题暂时无法应对，请尝试到论坛求助，我们会查看是否可以解决问题。

2、重用通过队列发送的缓冲区

请记住，每次将张量放入多进程队列 `multiprocessing.Queue` 时，它必须被移动到共享内存中。如果它已经被共享，将会是一个空操作，否则会产生一个额外的内存拷贝，这会减慢整个过程。即使您有一组进程将数据发送到单个进程，也可以让它将缓冲区发送回去，这几乎是不占资源的，并且可以在发送下一批时避免产生拷贝动作。

3、异步多进程训练（如：Hogwild）

使用多进程处理 `torch.multiprocessing`，可以异步地训练一个模型，参数既可以一直共享，也可以周期性同步。在第一种情况下，我们建议发送整个模型对象，而在后者中，我们建议只发送状态字典 `state_dict()`。

我们建议使用多进程处理队列 `multiprocessing.Queue` 在进程之间传递各种 PyTorch 对象。使用 `fork` 启动一个方法时，它也可能会继承共享内存中的张量和存储空间，但这种方式也非常容易出错，应谨慎使用，最好只能让高阶用户使用。而队列，尽管它们有时候不太优雅，却能在任何情况下正常工作。

警告：

你应该留意没有用 `if __name__ == '__main__':` 来保护的全局语句。如果使用了不同于 `fork` 启动方法，它们将在所有子进程中执行。

4、Hogwild

具体的 Hogwild 实现可以在 [示例库](#) 中找到，但为了展示代码的整体结构，下面还有一个最简单的示例：

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

def train(model):
    # 构建 data_loader, 优化器等
    for data, labels in data_loader:
        optimizer.zero_grad()
        loss_fn(model(data), labels).backward()
        optimizer.step() # 更新共享的参数

if __name__ == '__main__':
    num_processes = 4
    model = MyModel()
    # 注意: 这是 "fork" 方法工作所必需的
    model.share_memory()
    processes = []
    for rank in range(num_processes):
        p = mp.Process(target=train, args=(model,))
        p.start()
        processes.append(p)
    for p in processes:
        p.join()
```

译者署名

用户名	头像	职能	签名
风中劲草		翻译	人生总要追求点什么

序列化语义

最佳实践

保存模型的推荐方法

序列化和恢复模型有两种主要方法。

第一个（推荐）只保存和加载模型参数：

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

然后：

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

第二个方法是保存并加载整个模型：

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

然后：

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

然而，在这种情况下，序列化的数据会与特定的类结构和准确的目录结构相绑定，所以在其他项目中使用或经大量重构之后，这些结构可能会以各种方式被破坏。

Windows 常见问题

- 源码构建
 - 包含可选组件
 - 加速 Windows 的 CUDA 构建
 - 一个关键的安装脚本
- 扩展
 - CFFI 扩展
 - Cpp 扩展
- 安装
 - 在 win-32 中找不到包
 - 为什么没有 Windows 的 Python 2 包？
 - 导入错误
- 用法（多进程处理）
 - 无 if 语句保护的多进程处理错误
 - 多进程处理错误“损坏的管道”
 - 多进程处理错误“驱动程序关闭”
 - CUDA IPC 业务

源码构建

包含可选组件

Windows PyTorch 支持两种组件：MKL 和 MAGMA。以下是与他们一起构建的步骤。

```
# REM 确保您安装了7z和卷曲
# REM 下载 MKL 文件
curl https://s3.amazonaws.com/oss-ci-windows/mkl_2018.2.185.7z -k
-0
7z x -aoa mkl_2018.2.185.7z -omkl

# REM 下载 MAGMA 文件
# cuda90 / cuda91 也可在以下行中找到
set CUDA_PREFIX=cuda80
curl -k https://s3.amazonaws.com/oss-ci-windows/magma_%CUDA_PREFIX%_release_mkl_2018.2.185.7z -o magma.7z
7z x -aoa magma.7z -omagma

# REM 设置基本的环境变量
set "CMAKE_INCLUDE_PATH=%cd%\mkl\include"
set "LIB=%cd%\mkl\lib;%LIB%"
set "MAGMA_HOME=%cd%\magma"
```

加速 Windows 的 CUDA 构建

Visual Studio 目前不支持并行自定义任务。作为替代，我们可以使用 `Ninja` 来并行化 CUDA 构建任务。它只能通过输入几行代码来使用。

```
# REM 先安装 ninja
pip install ninja

# REM 将其设置为 cmake 生成器
set CMAKE_GENERATOR=Ninja
```

一个关键的安装脚本

你可以在这里查阅 [脚本](#)，它会给你指引。

扩展

CFFI 扩展

对 CFFI 扩展的支持是非常实验性的。在 Windows 下启用它通常有两个步骤。

首先，在扩展对象中指定其他库以更在 Windows 上构建。

```
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 libraries when necessary, like cudart
)
```

其次，对于“unresolved external symbol state caused by `extern THCState *state;`”由 `extern THCState *state` 引起的未解析的外部符号状态，将源代码从 C 更改为 C++。

下面列出了一个例子：

```
#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;
}
```

Cpp 扩展

与前一种相比，这种类型的扩展有更好的支持。但是，它仍然需要一些手动配置。首先，您应该打开**VS 2017**的**x86_x64**交叉工具链命令提示符。然后，您可以打开其中的**Git-Bash**。它通常位于 `C:\Program Files\Git\git-bash.exe` 中。最后，您可以开始编译过程。

安装

在 **win-32** 中找不到包


```
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 不适用于 32 位系统。请使用 Windows 和 Python 64 位版本。

为什么没有 Windows 的 Python 2 包？

因为它不够稳定。在我们正式发布之前需要解决一些问题。你可以自己构建它。

导入错误

```
from torch._C import *
```

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

这个问题是由于必要文件丢失造成的。实际上，我们几乎包含了除 VC2017 可再发行组件之外 PyTorch 所需的所有必要文件。您可以通过键入以下命令来解决此问题。

```
conda install -c peterjc123 vc vs2017_runtime
```

另一个可能的原因可能是您使用的是没有 NVIDIA 图形卡的 GPU 版本。请将您的 GPU 软件包替换为 CPU 软件包。

用法（多进程处理）

无 if 语句保护的多进程处理错误

RuntimeError:

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

在当前进程完成引导阶段之前，已尝试开始一个新进程

This probably means that you are **not** using `fork` to start your child processes **and** you have forgotten to use the proper idiom

m

in the main module:

这可能意味着你没有使用 `fork` 来启动你的子进程，并且你忘记了在主模块中使用正确的用法：

```
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.

如果程序不会被冻结以生成可执行文件，则可以省略 "`freeze_support()`" 行。

Windows 上 `multiprocessing` 多进程处理的实现不同，它使用 `spawn` 而不是 `fork`。所以我们必须用 if 条件语句来保护代码不被执行多次。将您的代码重构为以下结构：

```
import torch

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

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

多进程处理错误“损坏的管道”

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

```
BrokenPipeError: [Errno 32] Broken pipe
```

在父进程发送数据完成之前，子进程先结束就会发生此错误。这说明你的代码可能有问题。你可以通过将 `DataLoader` 的 `num_worker` 减为零来调试代码，看看问题是否仍然存在。

多进程处理错误“驱动程序关闭”

```
Couldn't open shared file mapping: <torch_14808_1591070686>, error code: <1455> at torch\lib\TH\THAllocator.c:154
```

```
[windows] driver shut down
```

请更新您的图形驱动程序。如果这种情况持续存在，这可能是由于您的显卡太旧或计算压力对于您的显卡来说太重了。请根据这篇 [文章](#) 更新 TDR 设置。

CUDA IPC 业务

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

Windows 不支持此类业务。就像在 CUDA 张量上进行多进程处理一样无法成功，有两种方法可供选择：

1. 不要使用多进程处理。将 `DataLoader` 的 `num_worker` 设置为零。
2. 改为 CPU 共享张量。确保您的自定义 `DataSet` 数据集返回 CPU 张量。

译者署名

用户名	头像	职能	签名
风中劲草		翻译	人生总要追求点什么

包参考

Torch

`torch` 包含了多维张量的数据结构以及基于其上的多种数学运算。此外，它也提供了多种实用工具，其中一些可以更有效地对张量和任意类型进行序列化的工具。

它具有CUDA的对应实现，可以在 `NVIDIA GPU` 上进行张量运算(计算能力 ≥ 3.0)。

张量 Tensors

```
torch.is_tensor(obj)
```

判断是否为张量，如果是pytorch张量，则返回True

- 参数： `obj (Object)` – 判断对象

```
torch.is_storage(obj)
```

判断是否为pytorch Storage，如果是，则返回True

- 参数： `input (Object)` – 判断对象

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

```
torch.numel(input)->int
```

返回 `input` 张量中的元素个数

- 参数: `input (Tensor)` – 输入张量

例子:

```
>>> 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)
```

设置打印选项。完全参考自 [Numpy](#)。

参数:

- `precision` – 浮点数输出的精度位数 (默认为8)
- `threshold` – 阈值, 触发汇总显示而不是完全显示(`repr`)的数组元素的总数 (默认为1000)
- `edgeitems` – 每个维度的开头和结尾的摘要中的数组项目数 (默认为3)。
- `linewidth` – 用于插入行间隔的每行字符数 (默认为80)。阈值矩阵将忽略此参数。
- `profile` – `pretty`打印的完全默认值。可以覆盖上述所有选项 (默认为`short, full`)

创建操作 **Creation Ops**

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

返回一个2维张量, 对角线数字为1, 其它位置为0

参数:

- `n` (int) – 行数
- `m` (int, 可选) – 列数. 如果为`None`, 则默认为`n`
- `out` (Tensor, 可选) - 输出张量

返回值: 2维张量, 对角线为1, 其它为0

返回类型: [Tensor](#)

例子:

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

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

将 `numpy.ndarray` 转换为 `Tensor` 。返回的张量`tensor`和`numpy`的`ndarray`共享同一内存空间。修改一个会导致另外一个也被修改。返回的张量不能调整大小。

例子:

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

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

返回`start`和`end`之间长度为 `steps` 的一维张量 参数:

- `start (float)` – 点集的起始值
- `end (float)` – 点集的最终值
- `steps (int)` – 在 `start` 和 `end` 间的采样数,即返回多少个数
- `out (Tensor, 可选的)` – 结果张量

例子:

```
>>> torch.linspace(1.0, 10.0, steps=5, out=None)
```

```
1.0000
3.2500
5.5000
7.7500
10.0000
[torch.FloatTensor of size 5]
```

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

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

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

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

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

返回一个1维张量，包含在区间 10^{start} 和 10^{end} 上以对数刻度均匀间隔的 `steps` 个点。输出1维张量的长度为 `steps`。

参数:

- `start (float)` – 该点集的起始点
- `end (float)` – 该点集的最终值
- `steps (int)` – 在 `start` 和 `end` 间生成的样本数
- `out (Tensor, 可选)` – 结果张量

例子:


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

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

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

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

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

返回一个全为1的张量，形状由可变参数 `sizes` 定义。

参数:

- `sizes (int...)` – 整数序列，定义了输出形状,如：`(5,5)`，`(2)`
- `out (Tensor, 可选)` – 结果张量

例子:

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

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

>>> torch.ones(5)

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

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

返回一个张量，填充在[0,1]区间的一组均匀分布随机数。Tensor的形状由变量 `sizes` 定义。

参数:

- `sizes (int...)` - 整数序列，定义了输出形状
- `out (Tensor, 可选)` - 结果张量

例子：

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

0.9193
0.3347
0.3232
0.7715
[torch.FloatTensor of size 4]

>>> torch.rand(2, 3, out=None)

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

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

返回一个张量，包含了从正态分布(均值为0，方差为 1，即高斯白噪声)中抽取一组随机数。Tensor的形状由变量 `sizes` 定义。

参数:

- `sizes (int...)` - 整数序列，定义了输出形状
- `out (Tensor, 可选)` - 结果张量

例子::

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

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

>>> torch.randn(2, 3)

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

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

输入参数 `n`，返回一个从 `0` 到 `n - 1` 的随机整数排列。

参数:

- `n(int)` – 上限(独占),即最大值

例子:

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

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

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

返回一个1维张量，长度为`floor((end-start)/step)`,`floor`代表向下取整。包含从 `start` 到 `end`，以 `step` 为步长的一组序列值(默认步长为1)。

参数:

- `start (float)` – 该点集的起始点
- `end (float)` – 该点集的终止点
- `step (float)` – 相邻点的间隔大小
- `out (Tensor, 可选的)` – 结果张量

例子:

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

1
2
3
[torch.FloatTensor of size 3]

>>> torch.arange(1, 2.5, 0.5)

1.0000
1.5000
2.0000
[torch.FloatTensor of size 3]
```

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

返回一个1维张量，长度为 $\text{floor}((\text{end}-\text{start})/\text{step})+1$ ，其中 floor 代表向下取整数。从 `start` 开始，`end` 为结尾，以 `step` 为步长的一组值。`step` 是两个值之间的间隔，即 $X_{i+1}=X_i+\text{step}$

参数:

- `start (float)` – 该点集的起始点
- `end (float)` – 该点集的最终值
- `step (int)` – 相邻点之间的间隔大小
- `out (Tensor, 可选的)` – 结果张量

例子：

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

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

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

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

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

返回一个全0的张量，形状由可变参数 `sizes` 定义。

参数:

- `sizes` (int...) – 整数序列，定义了输出形状
- `out` (Tensor, 可选) – 结果张量

例子：

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

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

>>> torch.zeros(5)

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

索引,切片,连接,变异操作

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

在给定维度上对输入的张量序列 `seq` 进行连接操作。

`torch.cat()` 可以看做 `torch.split()` 和 `torch.chunk()` 的逆运算。

`cat()` 函数可以通过下面例子很好的理解。

参数:

- `seq` (Tensors的序列) - 可以是相同类型的Tensor的任何python序列。
- `dim` (int, 可选) - 张量连接的尺寸
- `out` (Tensor, 可选) - 输出参数

例子：

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

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

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

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

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

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

```
```python
torch.chunk(tensor, chunks, dim=0)
```py
```

将张量沿着给定维度分解成的多个块。

参数：

- * tensor (Tensor) - 待分块的输入张量
- * chunks (int) - 分块的个数
- * dim (int) - 沿着此维度进行分块

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

沿给定轴`dim`，将输入索引张量`index`指定位置的值进行聚合。

对一个3维张量，输出可以定义为：

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

如果输入是一个n维张量 $(x_0, x_1, \dots, x_{i-1}, x_i, x_{i+1}, \dots, x_{n-1})$ 和暗=我，然后指数必须是一个n维张量的大小 $(x_0, x_1, \dots, x_{i-1}, y, x_{i+1}, \dots, x_{n-1})$ 其中 $y > = 1$ 和有同样大小的指标。

参数：

- * input(Tensor) - 源张量
- * dim(int) - 要索引的轴
- * index(LongTensor) - 要收集的元素的索引
- * out(Tensor, 可选) - 目的张量

例子：

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

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

返回一个新的张量，其索引input 张量沿尺寸 dim使用的条目中index这是一个Long Tensor。

返回的Tensor具有与原始Tensor相同数量的尺寸。

注意： 返回的张量不与原始张量共享内存空间。

参数：

- * input (Tensor) - 输入张量
- * dim (int) - 索引的轴
- * index (LongTensor) - 包含索引下标的一维张量
- * out (Tensor, 可选的) - 目标张量

例子：

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

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

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

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

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

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

```
* * *
```

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

根据掩码张量`mask`中的二元值，取输入张量中的指定项(`mask`为一个 `_ByteTensor`)，将取值返回到一个新的1D张量，

张量 `mask` 须跟`input`张量有相同数量的元素数目，但形状或维度不需要相同。
注意： 返回的张量不与原始张量共享内存空间。

参数：

- * `input (Tensor)` - 输入张量
- * `mask (ByteTensor)` - 掩码张量，包含了二元索引值
- * `out (Tensor, 可选的)` - 目标张量

例子：

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

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

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

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

```
||| torch.index_select(x, 1, indices)
```

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

```
### torch.nonzero
```

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

返回一个包含输入`input`中非零元素索引的张量。输出张量中的每行包含输入中非零元素的索引。

如果输入`input`有`n`维，则输出的索引张量`output`的形状为 $z \times n$ ，这里 z 是输入张量`input`中所有非零元素的个数。

参数：

- * `input` (Tensor) - 源张量
- * `out` (LongTensor, 可选的) - 包含索引值的结果张量

例子：

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

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

```
||| torch.nonzero(torch.Tensor([[0.6, 0.0, 0.0, 0.0], ... [0.0, 0.4, 0.0, 0.0],
... [0.0, 0.0, 1.2, 0.0], ... [0.0, 0.0, 0.0, -0.4]]))
```

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

```
### torch.split
```

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

将输入张量分割成相等形状的chunks（如果可分）。如果沿指定维的张量形状大小不能被`split_size`整分，则最后一个分块会小于其它分块。

参数：

- * `tensor` (Tensor) - 待分割张量
- * `split_size` (int) - 单个分块的形状大小
- * `dim` (int) - 沿着此维进行分割

```
### torch.squeeze
```

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

将输入张量形状中的`1`去除并返回。如果输入是形如 $((A \times 1 \times B \times 1 \times C \times 1 \times D))$ ，那么输出形状就为： $((A \times B \times C \times D))$

当给定`dim`时，那么挤压操作只在给定维度上。例如，输入形状为： $((A \times 1 \times B))$ ，`squeeze(input, 0)`将会保持张量不变，只有用`squeeze(input, 1)`，形状会变成 $((A \times B))$ 。

注意：返回张量与输入张量共享内存，所以改变其中一个的内容会改变另一个。

参数：

- * input (Tensor) - 输入张量
- * dim (int, 可选的) - 如果给定，则`input`只会在给定维度挤压
- * out (Tensor, 可选的) - 输出张量

例子：

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

```

## torch.stack[\[source\]](#)

```
torch.stack(sequence, dim=0)
```py

```

沿着一个新维度对输入张量序列进行连接。序列中所有的张量都应该为相同形状。

参数：

- * squence (Sequence) - 待连接的张量序列
- * dim (int) - 插入的维度。必须介于 0 与 待连接的张量序列数之间。

```
### torch.t
```

`torch.t(input, out=None) → Tensor`

输入一个矩阵（2维张量），并转置0, 1维。可以被视为函数`transpose(input, 0, 1)`的简写函数。

参数：

- * input (Tensor) – 输入张量
- * out (Tensor, 可选的) – 结果张量

```
>>> x = torch.randn(2, 3)
>>> x
0.4834  0.6907  1.3417
-0.1300  0.5295  0.2321
[torch.FloatTensor of size 2x3]
```py
```

```
>> torch.t(x)
```

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

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

返回输入矩阵 input 的转置。交换维度 dim0 和 dim1。输出张量与输入张量共享内存，所以改变其中一个会导致另外一个也被修改。

参数：

- input (Tensor) – 输入张量
- dim0 (int) – 转置的第一维
- dim1 (int) – 转置的第二维

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

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

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

 0.5983  1.5981
-0.0341 -0.5265
 2.4918 -0.8735
[torch.FloatTensor of size 3x2]
```

torch.unbind

```
torch.unbind(tensor, dim=0)[source]
```

移除指定维后，返回一个元组，包含了沿着指定维切片后的各个切片

参数:

- tensor (Tensor) – 输入张量
- dim (int) – 删除的维度

torch.unsqueeze

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

返回一个新的张量，对输入的制定位置插入维度 1

注意：返回张量与输入张量共享内存，所以改变其中一个的内容会改变另一个。

如果 dim 为负，则将会被转化(dim+input.dim()+1)

参数:

- tensor (Tensor) – 输入张量
- dim (int) – 插入维度的索引
- out (Tensor, 可选的) – 结果张量

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

随机抽样 **Random sampling**

torch.manual_seed

```
torch.manual_seed(seed)
```

设定生成随机数的种子，并返回一个 *torch._C.Generator* 对象。

参数: seed (int or long) – 种子。

torch.initial_seed

```
torch.initial_seed()
```

返回生成随机数的原始种子值 (python long) 。

torch.get_rng_state

```
torch.get_rng_state()[source]
```

返回随机生成器状态(*ByteTensor*)

torch.set_rng_state

```
torch.set_rng_state(new_state)[source]
```

设定随机生成器状态 参数: new_state (torch.ByteTensor) – 期望的状态

torch.default_generator

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

torch.bernoulli

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

从伯努利分布中抽取二元随机数(0 或者 1)。

输入张量须包含用于抽取上述二元随机值的概率。因此，输入中的所有值都必须在 $[0,1]$ 区间，即 $(0 \leq \text{input}_i \leq 1)$

输出张量的第 i 个元素值，将会以输入张量的第 i 个概率值等于 1 。

返回值将会是与输入相同大小的张量，每个值为0或者1 参数:

- input (Tensor) – 输入为伯努利分布的概率值
- out (Tensor, 可选的) – 输出张量(可选)

例子：

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

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

>>> torch.bernoulli(a)

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

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

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

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

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

torch.multinomial

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

返回一个张量，每行包含从 `input` 相应行中定义的多项分布中抽取的 `num_samples` 个样本。

当抽取样本时，依次从左到右排列(第一个样本对应第一列)。

如果输入 `input` 是一个向量，输出 `out` 也是一个相同长度 `num_samples` 的向量。如果输入 `input` 是有 `(m)` 行的矩阵，输出 `out` 是形如 `(m \times n)` 的矩阵。

如果参数 `replacement` 为 `True`，则样本抽取可以重复。否则，一个样本在每行不能被重复抽取。

参数 `num_samples` 必须小于 `input` 长度(即, `input` 的列数, 如果是 `input` 是一个矩阵)。

参数:

- `input` (Tensor) – 包含概率值的张量
- `num_samples` (int) – 抽取的样本数
- `replacement` (bool, 可选的) – 布尔值, 决定是否重复抽取
- `out` (Tensor, 可选的) – 结果张量

例子:

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

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

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

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

`torch.normal()`

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

返回一个张量, 包含从给定参数 `means`, `std` 的离散正态分布中抽取随机数。均值 `means` 是一个张量, 包含每个输出元素相关的正态分布的均值。`std` 是一个张量, 包含每个输出元素相关的正态分布的标准差。均值和标准差的形状不须匹配, 但每个张量的元素个数须相同。

参数:

- `means` (Tensor) – 均值
- `std` (Tensor) – 标准差
- `out` (Tensor) – 可选的输出张量


```
torch.normal(means=torch.arange(1, 11), std=torch.arange(1, 0, -0.1))

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

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

与上面函数类似，所有抽取的样本共享均值。

参数:

- means (Tensor, 可选的) – 所有分布均值
- std (Tensor) – 每个元素的标准差
- out (Tensor) – 可选的输出张量

例子:

```
>>> torch.normal(mean=0.5, std=torch.arange(1, 6))

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

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

与上面函数类似，所有抽取的样本共享标准差。

参数:

- means (Tensor) – 每个元素的均值
- std (float, 可选的) – 所有分布的标准差
- out (Tensor) – 可选的输出张量

例子:

```
>>> torch.normal(means=torch.arange(1, 6))

1.1681
2.8884
3.7718
2.5616
4.2500
[torch.FloatTensor of size 5]
```

序列化 **Serialization**

torch.save[\[source\]](#)

```
torch.save(obj, f, pickle_module=<module 'pickle' from '/home/jenkins/miniconda/lib/python3.5/pickle.py'>, pickle_protocol=2)
```

保存一个对象到一个硬盘文件上 参考: [Recommended approach for saving a model](#) 参数:

- **obj** – 保存对象
- **f** – 类文件对象 (返回文件描述符) 或一个保存文件名的字符串
- **pickle_module** – 用于pickling元数据和对象的模块
- **pickle_protocol** – 指定pickle protocol 可以覆盖默认参数

torch.load[\[source\]](#)

```
torch.load(f, map_location=None, pickle_module=<module 'pickle' from '/home/jenkins/miniconda/lib/python3.5/pickle.py'>)
```

从磁盘文件中读取一个通过 `torch.save()` 保存的对象。 `torch.load()` 可通过参数 `map_location` 动态地进行内存重映射, 使其能从不动设备中读取文件。一般调用时, 需两个参数: **storage** 和 **location tag**. 返回不同地址中的**storage**, 或着返回**None** (此时地址可以通过默认方法进行解析). 如果这个参数是字典的话, 意味着其是从文件的地址标记到当前系统的地址标记的映射。默认情况下, **location tags**中 "cpu"对应host tensors, 'cuda:device_id' (e.g. 'cuda:2') 对应cuda tensors。用户可以通过**register_package**进行扩展, 使用自己定义的标记和反序列化方法。

参数:

- **f** – 类文件对象 (返回文件描述符) 或一个保存文件名的字符串
- **map_location** – 一个函数或字典规定如何**remap**存储位置
- **pickle_module** – 用于unpickling元数据和对象的模块 (必须匹配序列化文件时

的pickle_module)

例子:

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

并行化 **Parallelism**

torch.get_num_threads

```
torch.get_num_threads() → int
```

获得用于并行化CPU操作的OpenMP线程数

torch.set_num_threads

```
torch.set_num_threads(int)
```

设定用于并行化CPU操作的OpenMP线程数

数学操作**Math operations**

Pointwise Ops

torch.abs

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

计算输入张量的每个元素绝对值

例子：

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

torch.acos(input, out=None) → Tensor

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

返回一个新张量，包含输入张量每个元素的反余弦。参数：

- input (Tensor) – 输入张量
- out (Tensor, 可选的) – 结果张量

例子：

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

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

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

torch.add()

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

对输入张量 `input` 逐元素加上标量值 `value`，并返回结果到一个新的张量 `out`，即 (`out = tensor + value`)。

如果输入 `input` 是 `FloatTensor` or `DoubleTensor` 类型，则 `value` 必须为实数，否则须为整数。

- input (Tensor) – 输入张量
- value (Number) – 添加到输入每个元素的数
- out (Tensor, 可选的) – 结果张量

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

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

>>> torch.add(a, 20)

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

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

`other` 张量的每个元素乘以一个标量值 `value`，并加到 `input` 张量上。返回结果到输出张量 `out`。即， $(out = input + (other * value))$

两个张量 `input` and `other` 的尺寸不需要匹配，但元素总数必须一样。

如果 `other` 是 `FloatTensor` or `DoubleTensor` 类型，则 `value` 必须为实数，否则须为整数。

参数:

- `input (Tensor)` – 第一个输入张量
- `value (Number)` – 用于第二个张量的尺寸因子
- `other (Tensor)` – 第二个输入张量
- `out (Tensor, 可选的)` – 结果张量

例子：

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

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

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

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

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

torch.addcddiv

```
torch.addcddiv(tensor, value=1, tensor1, tensor2, out=None) → Tensor
```

对 `tensor2` 对 `tensor1` 逐元素相除，然后乘以标量值 `value` 并加到 `tensor`。

张量的形状不需要匹配，但元素数量必须一致。

如果输入是 `FloatTensor` or `DoubleTensor` 类型，则 `value` 必须为实数，否则须为整数。

参数：

- `tensor (Tensor)` – 张量，对 `tensor1 ./ tensor2` 进行相加
- `value (Number, 可选的)` – 标量，对 `tensor1 ./ tensor2` 进行相乘
- `tensor1 (Tensor)` – 张量，作为被除数(分子)
- `tensor2 (Tensor)` – 张量，作为除数(分母)
- `out (Tensor, 可选的)` – 输出张量

例子：

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

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

torch.addcmul

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

用 `tensor2` 对 `tensor1` 逐元素相乘，并对结果乘以标量值 `value` 然后加到 `tensor`。张量的形状不需要匹配，但元素数量必须一致。如果输入是 `FloatTensor` or `DoubleTensor` 类型，则 `value` 必须为实数，否则须为整数。

参数：

- `tensor` (Tensor) – 张量，对 `tensor1 ./ tensor` 进行相加
- `value` (Number, 可选的) – 标量，对 `tensor1 . tensor2` 进行相乘
- `tensor1` (Tensor) – 张量，作为乘子1
- `tensor2` (Tensor) – 张量，作为乘子2
- `out` (Tensor, 可选的) – 输出张量

例子：

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

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

torch.asin

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

返回一个新张量，包含输入 `input` 张量每个元素的反正弦函数

参数：

- tensor (Tensor) – 输入张量
- out (Tensor, 可选的) – 输出张量

例子：

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

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

torch.atan

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

返回一个新张量，包含输入 `input` 张量每个元素的反正切函数

参数：

- tensor (Tensor) – 输入张量
- out (Tensor, 可选的) – 输出张量

例子：

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

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


torch.atan2

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

返回一个新张量，包含两个输入张量 `input1` 和 `input2` 的反正切函数

参数：

- `input1` (Tensor) – 第一个输入张量
- `input2` (Tensor) – 第二个输入张量
- `out` (Tensor, 可选的) – 输出张量

例子：

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

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

torch.ceil

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

天井函数，对输入 `input` 张量每个元素向上取整，即取不小于每个元素的最小整数，并返回结果到输出。

参数：

- `input` (Tensor) – 输入张量
- `out` (Tensor, 可选的) – 输出张量

例子：

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

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

>>> torch.ceil(a)

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

torch.clamp

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

将输入 `input` 张量每个元素的夹紧到区间 `([min, max])`，并返回结果到一个新张量。

操作定义如下：

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

如果输入是 `FloatTensor` or `DoubleTensor` 类型，则参数 `min` `max` 必须为实数，否则须为整数。

参数：

- `input (Tensor)` – 输入张量
- `min (Number)` – 限制范围下限
- `max (Number)` – 限制范围上限
- `out (Tensor, 可选的)` – 输出张量

例子：

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

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

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

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

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

将输入 `input` 张量每个元素的限制到不小于 `min`，并返回结果到一个新张量。

如果输入是 `FloatTensor` or `DoubleTensor` 类型，则参数 `value` 必须为实数，否则须为整数。

参数：

- `input (Tensor)` – 输入张量
- `value (Number)` – 限制范围下限
- `out (Tensor, 可选的)` – 输出张量

例子：

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

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

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

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

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

将输入 `input` 张量每个元素的限制到不大于 `max` ，并返回结果到一个新张量。

如果输入是 `FloatTensor` or `DoubleTensor` 类型，则参数 `value` 必须为实数，否则须为整数。

参数：

- `input (Tensor)` – 输入张量
- `value (Number)` – 限制范围上限
- `out (Tensor, 可选的)` – 输出张量

例子：

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

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

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

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

torch.cos

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

返回一个新张量，包含输入 `input` 张量每个元素的余弦。

参数：

- `input (Tensor)` – 输入张量
- `out (Tensor, 可选的)` – 输出张量

例子：

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

>>> torch.cos(a)
 0.8041
 0.9633
 0.9018
 0.2557
[torch.FloatTensor of size 4]
```

torch.cosh

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

返回一个新张量，包含输入 `input` 张量每个元素的双曲余弦。

参数：

- `input (Tensor)` – 输入张量
- `out (Tensor, 可选的)` – 输出张量

例子：

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

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

torch.div()

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

将 `input` 逐元素除以标量值 `value`，并返回结果到输出张量 `out`。即 (`out=tensor/value`)

如果输入是 `FloatTensor` or `DoubleTensor` 类型，则参数 `value` 必须为实数，否则须为整数。

参数：

- `input (Tensor)` – 输入张量
- `value (Number)` – 除数
- `out (Tensor, 可选的)` – 输出张量

例子：

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

-0.6147
-1.1237
-0.1604
-0.6853
 0.1063
[torch.FloatTensor of size 5]

>>> torch.div(a, 0.5)

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

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

两张量 `input` 和 `other` 逐元素相除，并将结果返回到输出。即，(`out_i= input_i / other_i`)

两张量形状不须匹配，但元素数须一致。

注意：当形状不匹配时，`input` 的形状作为输出张量的形状。

参数：

- `input (Tensor)` – 张量(分子)
- `other (Tensor)` – 张量(分母)

- out (Tensor, 可选的) – 输出张量

例子：

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

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

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

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

>>> torch.div(a, b)

-0.2062  0.5251  0.3229 -0.0684
-0.9528  1.8525  0.6320  1.9559
 0.3881 -3.8625 -0.0253  0.2099
-0.3473  0.6306  0.1666 -2.5381
[torch.FloatTensor of size 4x4]
```

torch.exp

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

返回一个新张量，包含输入 `input` 张量每个元素的指数。

参数：

- input (Tensor) – 输入张量
- out (Tensor, 可选的) – 输出张量

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

torch.floor

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

床函数: 返回一个新张量，包含输入 `input` 张量每个元素的`floor`，即不小于元素的最大整数。

参数：

- `input` (Tensor) – 输入张量
- `out` (Tensor, 可选的) – 输出张量

例子：

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

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

>>> torch.floor(a)

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

torch.fmod

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

计算除法余数。除数与被除数可能同时含有整数和浮点数。此时，余数的正负与被除数相同。

参数：

- `input` (Tensor) – 被除数
- `divisor` (Tensor or float) – 除数，一个数或与被除数相同类型的张量
- `out` (Tensor, 可选的) – 输出张量

例子：


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

参考: `torch.remainder()` , 计算逐元素余数, 相当于python 中的 % 操作符。

torch.frac

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

返回每个元素的分数部分。

例子:

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

torch.lerp

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

对两个张量以 `start` , `end` 做线性插值, 将结果返回到输出张量。

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

参数:

- `start (Tensor)` – 起始点张量
- `end (Tensor)` – 终止点张量
- `weight (float)` – 插值公式的weight
- `out (Tensor, 可选的)` – 结果张量

例子:

```
>>> start = torch.arange(1, 5)
>>> end = torch.Tensor(4).fill_(10)
>>> start

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

>>> end

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

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

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

torch.log

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

计算 `input` 的自然对数

参数：

- `input` (Tensor) – 输入张量
- `out` (Tensor, 可选的) – 输出张量

例子：

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

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

>>> torch.log(a)

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

torch.log1p

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

计算 $(\text{input} + 1)$ 的自然对数 ($y_i = \log(x_i + 1)$)

注意：对值比较小的输入，此函数比 `torch.log()` 更准确。

如果输入是 `FloatTensor` or `DoubleTensor` 类型，则 `value` 必须为实数，否则须为整数。

参数：

- `input (Tensor)` – 输入张量
- `out (Tensor, 可选的)` – 输出张量

例子：

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

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

>>> torch.log1p(a)

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

torch.mul

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

用标量值 `value` 乘以输入 `input` 的每个元素，并返回一个新的结果张量。(`out=tensor * value`)

如果输入是 `FloatTensor` or `DoubleTensor` 类型，则 `value` 必须为实数，否则须为整数。

参数：

- `input (Tensor)` – 输入张量
- `value (Number)` – 乘到每个元素的数
- `out (Tensor, 可选的)` – 输出张量

例子：

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

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

>>> torch.mul(a, 100)

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

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

两个张量 `input` , `other` 按元素进行相乘，并返回到输出张量。即计算(
`out_i=input_i * other_i`)

两计算张量形状不须匹配，但总元素数须一致。

参数：

- `input (Tensor)` – 第一个相乘张量
- `other (Tensor)` – 第二个相乘张量
- `out (Tensor, 可选的)` – 结果张量

例子：

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

-0.7280  0.0598 -1.4327 -0.5825
-0.1427 -0.0690  0.0821 -0.3270
-0.9241  0.5110  0.4070 -1.1188
-0.8308  0.7426 -0.6240 -1.1582
[torch.FloatTensor of size 4x4]

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

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

>>> torch.mul(a, b)

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

torch.neg

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

返回一个新张量，包含输入 `input` 张量按元素取负。即， $(out = -1 * input)$

参数：

- `input (Tensor)` – 输入张量
- `out (Tensor, 可选的)` – 输出张量

例子：

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

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

>>> torch.neg(a)

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

torch.pow

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

对输入 `input` 的按元素求 `exponent` 次幂值，并返回结果张量。幂值 `exponent` 可以为单一 `float` 数或者与 `input` 相同元素数的张量。

当幂值为标量时，执行操作： $out_i = x^{exponent}$

当幂值为张量时，执行操作： $out_i = x^{exponent_i}$

参数：

- `input` (Tensor) – 输入张量
- `exponent` (float or Tensor) – 幂值
- `out` (Tensor, 可选的) – 输出张量

例子：

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

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

>>> torch.pow(a, 2)

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

>>> exp = torch.arange(1, 5)
>>> a = torch.arange(1, 5)
>>> a

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

>>> exp

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

>>> torch.pow(a, exp)

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

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

`base` 为标量浮点值, `input` 为张量, 返回的输出张量 `out` 与输入张量相同形状。

执行操作为: $out_i = base^{input_i}$

参数：

- base (float) – 标量值，指数的底
- input (Tensor) – 幂值
- out (Tensor, 可选的) – 输出张量

例子：

```
>>> exp = torch.arange(1, 5)
>>> base = 2
>>> torch.pow(base, exp)

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

torch.reciprocal

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

返回一个新张量，包含输入 `input` 张量每个元素的倒数，即 $1.0/x$ 。

参数：

- input (Tensor) – 输入张量
- out (Tensor, 可选的) – 输出张量

例子：

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

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

>>> torch.reciprocal(a)

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

torch remainder

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

返回一个新张量，包含输入 `input` 张量每个元素的除法余数。除数与被除数可能同时包含整数或浮点数。余数与除数有相同的符号。

参数：

- `input` (Tensor) – 被除数
- `divisor` (Tensor or float) – 除数，一个数或者与除数相同大小的张量
- `out` (Tensor, 可选的) – 输出张量

例子：

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

torch.round

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

返回一个新张量，将输入 `input` 张量每个元素舍入到最近的整数。

参数：

- `input` (Tensor) – 输入张量
- `out` (Tensor, 可选的) – 输出张量

例子：

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

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

>>> torch.round(a)

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

torch.rsqrt

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

返回一个新张量，包含输入 `input` 张量每个元素的平方根倒数。

参数：

- `input` (Tensor) – 输入张量
- `out` (Tensor, 可选的) – 输出张量

例子：

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

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

>>> torch.rsqrt(a)

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

torch.sigmoid

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

返回一个新张量，包含输入 `input` 张量每个元素的sigmoid值。

参数：

- `input` (Tensor) – 输入张量
- `out` (Tensor, 可选的) – 输出张量

例子：

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

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

>>> torch.sigmoid(a)

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

torch.sign

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

符号函数：返回一个新张量，包含输入 `input` 张量每个元素的正负。

参数：

- `input` (Tensor) – 输入张量
- `out` (Tensor, 可选的) – 输出张量

例子：

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

>>> torch.sign(a)
-1
 1
 1
 1
[torch.FloatTensor of size 4]
```

torch.sin

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

返回一个新张量，包含输入 `input` 张量每个元素的正弦。

参数：

- `input (Tensor)` – 输入张量
- `out (Tensor, 可选的)` – 输出张量

例子：

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

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

torch.sinh

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

返回一个新张量，包含输入 `input` 张量每个元素的双曲正弦。

参数：

- `input (Tensor)` – 输入张量
- `out (Tensor, 可选的)` – 输出张量

例子：

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

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

torch.sqrt

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

返回一个新张量，包含输入 `input` 张量每个元素的平方根。

参数：

- `input (Tensor)` – 输入张量
- `out (Tensor, 可选的)` – 输出张量

例子：

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

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

>>> torch.sqrt(a)

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

torch.tan

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

返回一个新张量，包含输入 `input` 张量每个元素的正切。

参数：

- `input` (Tensor) – 输入张量
- `out` (Tensor, 可选的) – 输出张量

例子：

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

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

torch.tanh

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

返回一个新张量，包含输入 `input` 张量每个元素的双曲正切。

参数：

- `input (Tensor)` – 输入张量
- `out (Tensor, 可选的)` – 输出张量

例子：

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

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

torch.trunc

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

返回一个新张量，包含输入 `input` 张量每个元素的截断值(标量`x`的截断值是最接近其的整数，其比`x`更接近零。简而言之，有符号数的小数部分被舍弃)。

参数：

- `input (Tensor)` – 输入张量
- `out (Tensor, 可选的)` – 输出张量

例子：


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

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

>>> torch.trunc(a)

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

Reduction Ops

torch.cumprod

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

返回输入沿指定维度的累积积。例如，如果输入是一个N元向量，则结果也是一个N元向量，第 i 个输出元素值为 $(y_i = x_1 * x_2 * x_3 * \dots * x_i)$

参数：

- input (Tensor) – 输入张量
- dim (int) – 累积积操作的维度
- out (Tensor, 可选的) – 结果张量

例子：

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

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

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

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

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

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

torch.cumsum

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

返回输入沿指定维度的累积和。例如，如果输入是一个N元向量，则结果也是一个N元向量，第 `i` 个输出元素值为 ($y_i = x_1 + x_2 + x_3 + \dots + x_i$)

参数：

- `input (Tensor)` – 输入张量
- `dim (int)` – 累积和操作的维度
- `out (Tensor, 可选的)` – 结果张量

例子：

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

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

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

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

torch.dist

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

返回 (`input` - `other`) 的 `p` 范数。

参数：

- `input (Tensor)` – 输入张量

- other (Tensor) – 右侧输入张量
- p (float, 可选的) – 所计算的范数
- out (Tensor, 可选的) – 结果张量

例子：

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

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

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

 0.7782
-0.5185
 1.4106
-2.4063
[torch.FloatTensor of size 4]

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

torch.mean

```
torch.mean(input) → float
```

返回输入张量所有元素的均值。

参数：input (Tensor) – 输入张量

例子：

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

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

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

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

返回输入张量给定维度 `dim` 上每行的均值。

输出形状与输入相同，除了给定维度上为1。

参数：

- `input (Tensor)` – 输入张量
- `dim (int)` – the dimension to reduce
- `out (Tensor, 可选的)` – 结果张量

例子：

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

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

>>> torch.mean(a, 1)

-0.8545
 0.0997
 0.2464
-0.2157
[torch.FloatTensor of size 4x1]
```

torch.median

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

返回输入张量给定维度每行的中位数，同时返回一个包含中位数的索引的 `LongTensor`。

`dim` 值默认为输入张量的最后一维。输出形状与输入相同，除了给定维度上为1。

注意: 这个函数还没有在 `torch.cuda.Tensor` 中定义

参数：

- `input (Tensor)` – 输入张量
- `dim (int)` – 缩减的维度
- `values (Tensor, 可选的)` – 结果张量
- `indices (Tensor, 可选的)` – 返回的索引结果张量

```
>>> a

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

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

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

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

torch.mode

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

返回给定维 `dim` 上，每行的众数值。同时返回一个 `LongTensor`，包含众数取的索引。`dim` 值默认为输入张量的最后一维。

输出形状与输入相同，除了给定维度上为1。

注意: 这个函数还没有在 `torch.cuda.Tensor` 中定义

参数：

- `input (Tensor)` – 输入张量
- `dim (int)` – 缩减的维度
- `values (Tensor, 可选的)` – 结果张量
- `indices (Tensor, 可选的)` – 返回的索引张量

例子：

```
>>> a

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

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

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

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

torch.norm

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

返回输入张量 `input` 的 `p` 范数。

参数：

- `input (Tensor)` – 输入张量
- `p (float, 可选的)` – 范数计算中的幂指数值

例子：

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

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

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

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

返回输入张量给定维 `dim` 上每行的 `p` 范数。输出形状与输入相同，除了给定维度上为1。

参数：

- `input (Tensor)` – 输入张量
- `p (float)` – 范数计算中的幂指数值
- `dim (int)` – 缩减的维度
- `out (Tensor, 可选的)` – 结果张量

例子：


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

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

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

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

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

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

torch.prod

```
torch.prod(input) → float
```

返回输入张量 `input` 所有元素的积。

参数：input (Tensor) – 输入张量

例子：

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

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

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

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

返回输入张量给定维度上每行的积。输出形状与输入相同，除了给定维度上为1。

参数：

- input (Tensor) – 输入张量
- dim (int) – 缩减的维度
- out (Tensor, 可选的) – 结果张量

例子：

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

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

>>> torch.prod(a, 1)

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

torch.std

```
torch.std(input) → float
```

返回输入张量 `input` 所有元素的标准差。

参数：- input (Tensor) – 输入张量

例子：

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

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

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

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

返回输入张量给定维度上每行的标准差。输出形状与输入相同，除了给定维度上为1。

参数：

- input (Tensor) – 输入张量
- dim (int) – 缩减的维度
- out (Tensor, 可选的) – 结果张量

例子：

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

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

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

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

torch.sum

```
torch.sum(input) → float
```

返回输入张量 `input` 所有元素的和。

输出形状与输入相同，除了给定维度上为1。

参数：

- input (Tensor) – 输入张量

例子：

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

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

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

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

返回输入张量给定维度上每行的和。输出形状与输入相同，除了给定维度上为1。

参数：

- input (Tensor) – 输入张量
- dim (int) – 缩减的维度
- out (Tensor, 可选的) – 结果张量

例子：

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

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

>>> torch.sum(a, 1)

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

torch.var

```
torch.var(input) → float
```

返回输入张量所有元素的方差

输出形状与输入相同，除了给定维度上为1。

参数：

- input (Tensor) – 输入张量

例子：

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

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

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

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

返回输入张量给定维度上每行的方差。输出形状与输入相同，除了给定维度上为1。

参数：

- input (Tensor) – 输入张量
- dim (int) – the dimension to reduce
- out (Tensor, 可选的) – 结果张量 例子：

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

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

```
>> torch.var(a, 1)
```

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

```
## 比较操作 Comparison Ops
```

```
### torch.eq
```

```
```python
```

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

```
```py
```

比较元素相等性。第二个参数可为一个数或与第一个参数同类型形状的张量。

参数：

* input (Tensor) - 待比较张量

* other (Tensor or float) - 比较张量或数

* out (Tensor, 可选的) - 输出张量，须为 ByteTensor类型 or 与`input`同类型

返回值： 一个 `torch.ByteTensor` 张量，包含了每个位置的比较结果(相等为1，不等为0)

返回类型： Tensor

例子：

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

torch.equal

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

```
```py
```

如果两个张量有相同的形状和元素值，则返回`True`，否则`False`。

例子：

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

## torch.ge

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

逐元素比较`input`和`other`，即是否 $(input \geq other)$ 。

如果两个张量有相同的形状和元素值，则返回`True`，否则`False`。第二个参数可以为一个数或与第一个参数相同形状和类型的张量

参数：

- * `input` (Tensor) - 待对比的张量
- * `other` (Tensor or float) - 对比的张量或`float`值
- * `out` (Tensor, 可选的) - 输出张量。必须为`ByteTensor`或者与第一个参数`tensor`相同类型。

返回值：一个`torch.ByteTensor`张量，包含了每个位置的比较结果(是否 $input \geq other$)。返回类型：Tensor

例子：

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

torch.gt

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

逐元素比较`input`和`other`，即是否  $(input > other)$ 。如果两个张量有相同的形状和元素值，则返回`True`，否则`False`。第二个参数可以为一个数或与第一个参数相同形状和类型的张量

参数：

- \* `input` (Tensor) - 要对比的张量
- \* `other` (Tensor or float) - 要对比的张量或`float`值
- \* `out` (Tensor, 可选的) - 输出张量。必须为`ByteTensor`或者与第一个参数`tensor`相同类型。

返回值：一个`torch.ByteTensor`张量，包含了每个位置的比较结果(是否  $input > other$ )。返回类型：Tensor

例子：

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

## torch.kthvalue

```
torch.kthvalue(input, k, dim=None, out=None) -> (Tensor, LongTensor)
```py

```

取输入张量`input`指定维上第`k`个最小值。如果不指定`dim`，则默认为`input`的最后一维。

返回一个元组 `_(values, indices)_`，其中`indices`是原始输入张量`input`中沿`dim`维的第`k`个最小值下标。

参数：

- * `input (Tensor)` - 要对比的张量
- * `k (int)` - 第`k`个最小值
- * `dim (int, 可选的)` - 沿着此维进行排序
- * `out (tuple, 可选的)` - 输出元组 (Tensor, LongTensor) 可选地给定作为输出 buffers

例子：

```
||| x = torch.arange(1, 6) x
```

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

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

torch.le

```
torch.le(input, other, out=None) -> Tensor
```py

```

逐元素比较`input`和`other`，即是否 `( input <= other )` 第二个参数可以为一个数或与第一个参数相同形状和类型的张量

参数：

- \* `input (Tensor)` - 要对比的张量
- \* `other (Tensor or float)` - 对比的张量或`float`值
- \* `out (Tensor, 可选的)` - 输出张量。必须为`ByteTensor`或者与第一个参数`tensor`相同类型。

返回值：一个`torch.ByteTensor`张量，包含了每个位置的比较结果(是否 `input >= other`)。返回类型：Tensor

例子：



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

## torch.lt

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

逐元素比较`input`和`other`，即是否  $(input < other)$

第二个参数可以为一个数或与第一个参数相同形状和类型的张量

参数：

- \* `input` (Tensor) - 要对比的张量
- \* `other` (Tensor or float) - 对比的张量或`float`值
- \* `out` (Tensor, 可选的) - 输出张量。必须为`ByteTensor`或者与第一个参数`tensor`相同类型。

`input`：一个`torch.ByteTensor`张量，包含了每个位置的比较结果(是否  $tensor \geq other$ )。返回类型：Tensor

例子：

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

## torch.max

```
torch.max()
``py
```

返回输入张量所有元素的最大值。

参数：

- \* `input` (Tensor) - 输入张量

例子：

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

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

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

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

```

返回输入张量给定维度上每行的最大值，并同时返回每个最大值的位置索引。

输出形状中，将`dim`维设定为1，其它与输入形状保持一致。

参数：

- * `input` (Tensor) - 输入张量
- * `dim` (int) - 指定的维度
- * `max` (Tensor, 可选的) - 结果张量，包含给定维度上的最大值
- * `max_indices` (LongTensor, 可选的) - 结果张量，包含给定维度上每个最大值的位置索引

例子：

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

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

```
torch.max(a, 1) ( 1.2513 0.9288 1.0695 0.7426 [torch.FloatTensor of
size 4x1] , 2 0 0 0 [torch.LongTensor of size 4x1] ) ```py

```

```
torch.max(input, other, out=None) -> Tensor
```py

```

返回输入张量给定维度上每行的最大值，并同时返回每个最大值的位置索引。即，`\(out_i=max(input_i,other_i) \)`

输出形状中，将`dim`维设定为1，其它与输入形状保持一致。

参数：

- \* `input` (Tensor) - 输入张量
- \* `other` (Tensor) - 输出张量
- \* `out` (Tensor, 可选的) - 结果张量

例子：

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

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

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

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

```
||| torch.max(a, b)
```

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

```
torch.min
```

`torch.min(input) → float`

返回输入张量所有元素的最小值。

参数: `input (Tensor)` - 输入张量

例子:

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

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

```
||| torch.min(a) -0.22663167119026184 ``py
```

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

返回输入张量给定维度上每行的最小值，并同时返回每个最小值的位置索引。

输出形状中，将`dim`维设定为1，其它与输入形状保持一致。

参数:

- \* `input (Tensor)` - 输入张量
- \* `dim (int)` - 指定的维度
- \* `min (Tensor, 可选的)` - 结果张量，包含给定维度上的最小值
- \* `min_indices (LongTensor, 可选的)` - 结果张量，包含给定维度上每个最小值的位置索引

例子:

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

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

```
||| torch.min(a, 1)
```

```
0.54
```

译者署名

用户名	头像	职能	签名
Song		翻译	人生总要追求点什么

...

## torch.Tensor

`torch.Tensor` 是一种包含单一数据类型元素的多维矩阵。

Torch定义了七种CPU张量类型和八种GPU张量类型：

Data tyoe	CPU tensor	GPU tensor
32-bit floating point	<code>torch.FloatTensor</code>	<code>torch.cuda.FloatTensor</code>
64-bit floating point	<code>torch.DoubleTensor</code>	<code>torch.cuda.DoubleTensor</code>
16-bit floating point	N/A	<code>torch.cuda.HalfTensor</code>
8-bit integer (unsigned)	<code>torch.ByteTensor</code>	<code>torch.cuda.ByteTensor</code>
8-bit integer (signed)	<code>torch.CharTensor</code>	<code>torch.cuda.CharTensor</code>
16-bit integer (signed)	<code>torch.ShortTensor</code>	<code>torch.cuda.ShortTensor</code>
32-bit integer (signed)	<code>torch.IntTensor</code>	<code>torch.cuda.IntTensor</code>
64-bit integer (signed)	<code>torch.LongTensor</code>	<code>torch.cuda.LongTensor</code>

`torch.Tensor` 是默认的tensor类型（`torch.FlaotTensor`）的简称。

张量可以从Python的 `list` 或序列构成：

```
>>> torch.FloatTensor([[1, 2, 3], [4, 5, 6]])
1 2 3
4 5 6
[torch.FloatTensor of size 2x3]
```

可以通过指定它的大小来构建一个空的张量：

```
>>> torch.IntTensor(2, 4).zero_()
0 0 0 0
0 0 0 0
[torch.IntTensor of size 2x4]
```

可以使用Python的索引和切片符号来访问和修改张量的内容：

```
>>> x = torch.FloatTensor([[1, 2, 3], [4, 5, 6]])
>>> print(x[1][2])
6.0
>>> x[0][1] = 8
>>> print(x)
 1 8 3
 4 5 6
[torch.FloatTensor of size 2x3]
```

每一个张量`tensor`都有一个相应的 `torch.Storage` 保存其数据。张量类提供了一个多维的、横向视图的存储，并定义了数字操作。

注意：改变张量的方法可以用一个下划线后缀来标示。比如，`torch.FloatTensor.abs_()` 会在原地计算绝对值并返回修改的张量，而 `torch.FloatTensor.abs()` 将会在新张量中计算结果。

```
class torch.Tensor
class torch.Tensor(*sizes)
class torch.Tensor(size)
class torch.Tensor(sequence)
class torch.Tensor(ndarray)
class torch.Tensor(tensor)
class torch.Tensor(storage)
```

从可选的大小或数据创建新的张量。如果没有给出参数，则返回空的零维张量。如果提供了 `numpy.ndarray`，`torch.Tensor` 或 `torch.Storage`，将会返回一个相同数据的新张量。如果提供了python序列，则从序列的副本创建一个新的张量。

## abs()

参考 `torch.abs()` ,矩阵数组绝对值

## abs\_()

`abs_()` 的直接运算形式

## acos()

参考 `torch.acos()`

## acos\_()

`acos_()` 的直接运算形式，即直接执行并且返回修改后的张量

## **add(value)**

参考 `torch.add()`

## **add\_(value)**

`add()` 的直接运算形式，即直接执行并且返回修改后的张量

## **addbmm(beta=1, mat, alpha=1, batch1, batch2)**

参考 `torch.addbmm()`

## **addbmm\_(beta=1, mat, alpha=1, batch1, batch2)**

`addbmm()` 的直接运算形式，即直接执行并且返回修改后的张量

## **addcdiv(value=1, tensor1, tensor2)**

参考 `torch.addcdiv()`

## **addcdiv\_(value=1, tensor1, tensor2)**

`addcdiv()` 的直接运算形式，即直接执行并且返回修改后的张量

## **addcmul(value=1, tensor1, tensor2)**

参考 `torch.addcmul()`

## **addcmul\_(value=1, tensor1, tensor2)**

`addcmul()` 的直接运算形式，即直接执行并且返回修改后的张量

## **addmm(beta=1, mat, alpha=1, mat1, mat2)**

参考 `torch.addmm()`

## **addmm\_(beta=1, mat, alpha=1, mat1, mat2)**

`addmm()` 的直接运算形式，即直接执行并且返回修改后的张量

## **addmv(beta=1, tensor, alpha=1, mat, vec)**

参考 `torch.addmv()`

## **addmv\_(beta=1, tensor, alpha=1, mat, vec)**

`addmv()` 的直接运算形式，即直接执行并且返回修改后的张量

## **addr(beta=1, alpha=1, vec1, vec2)**

参考 `torch.addr()`

## **addr\_(beta=1, alpha=1, vec1, vec2)**

`addr()` 的直接运算形式，即直接执行并且返回修改后的张量

## **apply\_(callable)**

将函数 `callable` 作用于 `tensor` 中每一个元素，并替换每个元素用 `callable` 函数返回的值。

注意：该函数只能在 CPU `tensor` 中使用，并且不应该用在有较高性能要求的代码块。

## **asin()**

参考 `torch.asin()`

## **asin\_()**

`asin()` 的直接运算形式，即直接执行并且返回修改后的张量

## **atan()**

参考 `torch.atan()`

## **atan2()**

参考 `torch.atan2()`

## **atan2\_()**

`atan2()` 的直接运算形式，即直接执行并且返回修改后的张量

## **atan\_()**

`atan()` 的直接运算形式，即直接执行并且返回修改后的张量



## **baddbmm(beta=1, alpha=1, batch1, batch2)**

参考 `torch.baddbmm()`

## **baddbmm\_(beta=1, alpha=1, batch1, batch2)**

`baddbmm()` 的直接运算形式，即直接执行并且返回修改后的张量

## **bernoulli()**

参考 `torch.bernoulli()`

## **bernoulli\_()**

`bernoulli()` 的直接运算形式，即直接执行并且返回修改后的张量

## **bmm(batch2)**

参考 `torch.bmm()`

## **byte()**

将tensor改为byte类型

## **bmm(median=0, sigma=1, \*, generator=None)**

将tensor中元素用柯西分布得到的数值填充： $P(x)=1/\pi * \sigma/(x-\text{median})^2+\sigma^2$

## **ceil()**

参考 `torch.ceil()`

## **ceil\_()**

`ceil()` 的直接运算形式，即直接执行并且返回修改后的张量

## **char()**

将tensor元素改为char类型

## **chunk(n\_chunks, dim=0)**

将tensor分割为tensor元组. 参考 `torch.chunk()`

## clamp(min, max)

参考 `torch.clamp()`

## clamp\_(min, max)

`clamp()` 的直接运算形式，即直接执行并且返回修改后的张量

## clone()

返回与原tensor有相同大小和数据类型的tensor

## contiguous()

返回一个内存连续的有相同数据的tensor，如果原tensor内存连续则返回原tensor

## copy\_(src, async=False)

将 `src` 中的元素复制到tensor中并返回这个tensor。如果**broadcast**是True，则源张量必须可以使用该张量广播。否则两个tensor应该有相同数目的元素，可以是不同的数据类型或存储在不同的设备上。

参数：

- **src** (Tensor) - 要复制的源张量
- **async** (bool) - 如果为True，并且此副本位于CPU和GPU之间，则副本可能会相对于主机异步发生。对于其他副本，此参数无效。
- **broadcast** (bool) - 如果为True，src将广播到底层张量的形状。

## cos()

参考 `torch.cos()`

## cos\_()

`cos()` 的直接运算形式，即直接执行并且返回修改后的张量

## cosh()

参考 `torch.cosh()`

## cosh\_()

`cosh()` 的直接运算形式，即直接执行并且返回修改后的张量

## cpu()

如果在CPU上没有该tensor，则会返回一个CPU的副本

## cross(other, dim=-1)

参考 `torch.cross()`

## cuda(device=None, async=False)

返回此对象在CPU内存中的一个副本 如果该对象已经在CUDA内存中，并且在正确的设备上，则不会执行任何副本，并返回原始对象。

参数：

- `device (int)` ：目标GPU ID。默认为当前设备。
- `async (bool)` ：如果为True并且源处于固定内存中，则该副本将相对于主机是异步的。否则，该参数没有意义。

## cumprod(dim)

参考 `torch.cumprod()`

## cumsum(dim)

参考 `torch.cumsum()`

## data\_ptr() → int

返回tensor第一个元素的地址

## diag(diagonal=0)

参考 `torch.diag()`

## dim() → int

返回tensor的维数

## dist(other, p=2)

参考 `torch.dist()`

## div(value)

参考 `torch.div()`

## **div\_(value)**

`div()` 的直接运算形式，即直接执行并且返回修改后的张量

## **dot(tensor2) → float**

参考 `torch.dot()`

## **double()**

将该tensor投射为double类型

## **eig(eigenvectors=False) -> (Tensor, Tensor)**

参考 `torch.eig()`

## **element\_size() → int**

返回单个元素的字节大小。例：

```
>>> torch.FloatTensor().element_size()
4
>>> torch.ByteTensor().element_size()
1
```

## **eq(other)**

参考 `torch.eq()`

## **eq\_(other)**

`eq()` 的直接运算形式，即直接执行并且返回修改后的张量

## **equal(other) → bool**

参考 `torch.equal()`

## **exp()**

参考 `torch.exp()`

## exp\_()

`exp()` 的直接运算形式，即直接执行并且返回修改后的张量

## expand(\*sizes)

返回tensor的一个新视图，单个维度扩大为更大的尺寸。`tensor`也可以扩大为更高维，新增加的维度将附在前面。扩大`tensor`不需要分配新内存，只是仅仅新建一个`tensor`的视图，其中通过将 `stride` 设为0，一维将会扩展位更高维。任何一个一维的在不分配新内存情况下可扩展为任意的数值。

参数：

- `sizes(torch.Size or int...)`-需要扩展的大小

例：

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

## expand\_as(tensor)

将`tensor`扩展为参数`tensor`的大小。该操作等效与：

```
self.expand(tensor.size())
```

## exponential\_(lambd=1, \*, generator=None) \$to\$ Tensor

将该`tensor`用指数分布得到的元素填充： $P(x) = \lambda e^{-\lambda x}$

## fill\_(value)

将该`tensor`用指定的数值填充

## float()

将`tensor`投射为`float`类型

## floor()

参考 `torch.floor()`

## floor\_()

`floor()` 的直接运算形式，即直接执行并且返回修改后的张量

## fmod(divisor)

参考 `torch.fmod()`

## fmod\_(divisor)

`fmod()` 的直接运算形式，即直接执行并且返回修改后的张量

## frac()

参考 `torch.frac()`

## frac\_()

`frac()` 的直接运算形式，即直接执行并且返回修改后的张量

## gather(dim, index)

参考 `torch.gather()`

## ge(other)

参考 `torch.ge()`

## ge\_(other)

`ge()` 的直接运算形式，即直接执行并且返回修改后的张量

## gels(A)

参考 `torch.gels()`

## geometric\_(p, \*, generator=None)

将该tensor用几何分布得到的元素填充：  $P(X=k) = (1-p)^{k-1}p$

## geqrf() -> (Tensor, Tensor)

参考 `torch.geqrf()`

## ger(vec2)

参考 `torch.ger()`

## gesv(A), Tensor

参考 `torch.gesv()`

## gt(other)

参考 `torch.gt()`

## gt\_(other)

`gt()` 的直接运算形式，即直接执行并且返回修改后的张量

## half()

将tensor投射为半精度浮点类型

## histc(bins=100, min=0, max=0)

参考 `torch.histc()`

## index(m)

用一个二进制的掩码或沿着一个给定的维度从tensor中选取元素。`tensor.index(m)` 与 `tensor[m]` 完全相同。

参数：

- `m(int or Byte Tensor or slice)`-用来选取元素的维度或掩码

## indexadd(dim, index, tensor)

按参数index中的索引数确定的顺序，将参数tensor中的元素加到原来的tensor中。参数tensor的尺寸必须严格地与原tensor匹配，否则会发生错误。

参数：

- `dim(int)`-索引index所指向的维度

- `index(LongTensor)`- 需要从`tensor`中选取的指数
- `tensor(Tensor)`- 含有相加元素的`tensor`

例：

```
>>> x = torch.Tensor([[1, 1, 1], [1, 1, 1], [1, 1, 1]])
>>> t = torch.Tensor([[1, 2, 3], [4, 5, 6], [7, 8, 9]])
>>> index = torch.LongTensor([0, 2, 1])
>>> x.index_add_(0, index, t)
>>> x
 2 3 4
 8 9 10
 5 6 7
[torch.FloatTensor of size 3x3]
```

## `indexcopy(dim, index, tensor)`

按参数`index`中的索引数确定的顺序，将参数`tensor`中的元素复制到原来的`tensor`中。参数`tensor`的尺寸必须严格地与原`tensor`匹配，否则会发生错误。

参数：

- `dim (int)`- 索引`index`所指向的维度
- `index (LongTensor)`- 需要从`tensor`中选取的指数
- `tensor (Tensor)`- 含有被复制元素的`tensor`

例：

```
>>> x = torch.Tensor(3, 3)
>>> t = torch.Tensor([[1, 2, 3], [4, 5, 6], [7, 8, 9]])
>>> index = torch.LongTensor([0, 2, 1])
>>> x.index_copy_(0, index, t)
>>> x
 1 2 3
 7 8 9
 4 5 6
[torch.FloatTensor of size 3x3]
```

## `indexfill(dim, index, val)`

按参数`index`中的索引数确定的顺序，将原`tensor`用参数 `val` 值填充。

参数：

- `dim (int)`- 索引`index`所指向的维度
- `index (LongTensor)`- 索引
- `val (Tensor)`- 填充的值

例：



```
>>> x = torch.Tensor([[1, 2, 3], [4, 5, 6], [7, 8, 9]])
>>> index = torch.LongTensor([0, 2])
>>> x.index_fill_(0, index, -1)
>>> x
 -1 2 -1
 -1 5 -1
 -1 8 -1
[torch.FloatTensor of size 3x3]
```

## index\_select(dim, index)

参考 `torch.index_select()`

## int()

将该tensor投射为int类型

## inverse()

参考 `torch.inverse()`

## is\_contiguous() → bool

如果该tensor在内存中是连续的则返回True。

## is\_cuda

## is\_pinned()

如果该tensor在固定内存中则返回True

## is\_set\_to(tensor) → bool

如果此对象引用与Torch C API相同的 `THTensor` 对象作为给定的张量，则返回True。

## is\_signed()

## kthvalue(k, dim=None) -> (Tensor, LongTensor)

参考 `torch.kthvalue()`

## le(other)

参考 `torch.le()`

## le\_(other)

`le()` 的直接运算形式，即直接执行并且返回修改后的张量

## lerp(start, end, weight)

参考 `torch.lerp()`

## lerp(\_start, end, weight)

`lerp()` 的直接运算形式，即直接执行并且返回修改后的张量

## log()

参考 `torch.log()`

## loglp()

参考 `torch.loglp()`

## loglp\_()

`loglp()` 的直接运算形式，即直接执行并且返回修改后的张量

## log\_()→ Tensor

`log()` 的直接运算形式，即直接执行并且返回修改后的张量

## lognormal(mwan=1, std=2, , gegnerator=None\*)

将该tensor用均值为 $\mu$ ,标准差为 $\sigma$ 的对数正态分布得到的元素填充。要注意 `mean` 和 `stdv` 是基本正态分布的均值和标准差，不是返回的分布：
$$P(X)=\frac{1}{x\sigma\sqrt{2\pi}}e^{-\frac{(\ln x-\mu)^2}{2\sigma^2}}$$

## long()

将tensor投射为long类型

## lt(other)

参考 `torch.lt()`

## **`lt(_other)`**

`lt()` 的直接运算形式，即直接执行并且返回修改后的张量

## **`map(_tensor, callable)`**

将 `callable` 作用于本`tensor`和参数`tensor`中的每一个元素，并将结果存放在本`tensor`中。`callable` 应该有下列标志：

```
def callable(a, b) -> number
```

## **`maskedcopy(mask, source)`**

将 `mask` 中值为1元素对应的 `source` 中位置的元素复制到本`tensor`中。`mask` 应该有和本`tensor`相同数目的元素。`source` 中元素的个数最少为 `mask` 中值为1的元素的个数。

参数：

- `mask (ByteTensor)`-二进制掩码
- `source (Tensor)`-复制的源`tensor`

注意：`mask` 作用于 `self` 自身的`tensor`，而不是参数中的 `source`。

## **`maskedfill(mask, value)`**

在 `mask` 值为1的位置处用 `value` 填充。`mask` 的元素个数需和本`tensor`相同，但尺寸可以不同。形状`mask`必须 与下面的张量的形状一起广播。

参数：

- `mask (ByteTensor)`-二进制掩码
- `value (Tensor)`-用来填充的值

## **`maskedselect(_mask)`**

参考 `torch.masked_select()`

## **`max(dim=None) -> float or(Tensor, Tensor)`**

参考 `torch.max()`

## **`mean(dim=None) -> float or(Tensor, Tensor)`**

参考 `torch.mean()`

**`median(dim=-1, value=None, indices=None) -> (Tensor, LongTensor)`**

参考 `torch.median()`

**`min(dim=None) -> float or(Tensor, Tensor)`**

参考 `torch.min()`

**`mm(mat2)`**

参考 `torch.mm()`

**`mode(dim=-1, value=None, indices=None) -> (Tensor, LongTensor)`**

参考 `torch.mode()`

**`mul(value)`**

参考 `torch.mul()`

**`mul(_value)`**

`mul()` 的直接运算形式，即直接执行并且返回修改后的张量

**`multinomial(num_samples, replacement=False, , generator=None*)`**

参考 `torch.multinomial()`

**`mv(vec)`**

参考 `torch.mv()`

**`narrow(dimension, start, length) → Te`**

返回这个张量的缩小版本的新张量。维度 `dim` 缩小范围是 `start` 到 `start+length`。返回的张量和该张量共享相同的底层存储。

参数：

- dimension (*int*)-需要缩小的维度
- start (*int*)-起始维度
- length (*int*)-长度

例:

```
>>> x = torch.Tensor([[1, 2, 3], [4, 5, 6], [7, 8, 9]])
>>> x.narrow(0, 0, 2)
 1 2 3
 4 5 6
[torch.FloatTensor of size 2x3]
>>> x.narrow(1, 1, 2)
 2 3
 5 6
 8 9
[torch.FloatTensor of size 3x2]
```

## **ndimension() → int**

`dim()` 的另一种表示。

## **ne(*other*)**

参考 `torch.ne()`

## **ne(*\_other*)**

`ne()` 的直接运算形式，即直接执行并且返回修改后的张量

## **neg()**

参考 `torch.neg()`

## **neg\_()**

`neg()` 的直接运算形式，即直接执行并且返回修改后的张量

## **nelement() → int**

`numel()` 的另一种表示

## **new(*args*, \**kwargs*)**

构建一个有相同数据类型的tensor

## **nonzero() → LongTensor**

参考`torch.nonzero()`

## **norm(*p*=2) → float**

参考`torch.norm()`

## **normal(*\_mean*=0, *std*=1, , *generator*=None\*)**

将tensor用均值为 `mean` 和标准差为 `std` 的正态分布填充。

## **numel() → int**

参考 `numel()`

## **numpy() → ndarray**

将该tensor以NumPy的形式返回 `ndarray`，两者共享相同的底层内存。原tensor改变后会相应的在 `ndarray` 有反映，反之亦然。

## **orgqr(*input2*)**

参考 `torch.orgqr()`

## **ormqr(*input2*, *input3*, *left*=True, *transpose*=False)**

参考 `torch.ormqr()`

## **permute(*dims*)**

将tensor的维度换位。

参数：

- `_dims` (`_int.*`)-换位顺序

例：

```
>>> 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()**

将张量复制到固定内存（如果尚未固定）。

## **potrf(*upper=True*)**

参考 `torch.potrf()`

## **potri(*upper=True*)**

参考 `torch.potri()`

## **potrs(*input2, upper=True*)**

参考 `torch.potrs()`

## **pow(*exponent*)**

参考 `torch.pow()`

## **pow\_()**

`pow()` 的直接运算形式，即直接执行并且返回修改后的张量

## **prod()) → float**

参考 `torch.prod()`

## **pstrf(*upper=True, tol=-1*) -> (Tensor, IntTensor)**

参考 `torch.pstrf()`

## **qr()-> (Tensor, IntTensor)**

参考 `torch.qr()`

## **random\_(*from=0, to=None, \*, generator=None*)**

将tensor用从在[*from*, *to*-1]上的正态分布或离散正态分布取样值进行填充。如果未指定，则该值仅由该张量数据类型限定。

## **reciprocal()**

参考 `torch.reciprocal()`

## **`reciprocal_()`**

`reciprocal()` 的直接运算形式，即直接执行并且返回修改后的张量

## **`remainder(divisor)`**

参考 `torch.remainder()`

## **`remainder(_divisor)`**

`remainder()` 的直接运算形式，即直接执行并且返回修改后的张量

## **`renorm(p, dim, maxnorm)`**

参考 `torch.renorm()`

## **`renorm(_p, dim, maxnorm)`**

`renorm()` 的直接运算形式，即直接执行并且返回修改后的张量

## **`repeat(*sizes)`**

沿指定维度重复此张量。与 `expand()` 功能不同，此功能可复制张量中的数据。

参数：

- *\*sizes* (`torch.Size` or `int...`)-沿着每个维重复这个张量的次数

例：

```
>>> x = torch.Tensor([1, 2, 3])
>>> x.repeat(4, 2)
 1 2 3 1 2 3
 1 2 3 1 2 3
 1 2 3 1 2 3
 1 2 3 1 2 3
[torch.FloatTensor of size 4x6]
>>> x.repeat(4, 2, 1).size()
torch.Size([4, 2, 3])
```

## **`resize_(*sizes)`**



将**tensor**的大小调整为指定的大小。如果元素个数比当前的内存大小大，就将底层存储大小调整为与新元素数目一致的大小。如果元素个数比当前内存小，则底层存储不会被改变。原来**tensor**中被保存下来的元素将保持不变，但新内存将不会被初始化。

参数：

- **sizes** (**torch.Size** or **int...**)-需要调整的大小

例：

```
>>> x = torch.Tensor([[1, 2], [3, 4], [5, 6]])
>>> x.resize_(2, 2)
>>> x
 1 2
 3 4
[torch.FloatTensor of size 2x2]
```

## resizeas(tensor)

将当前张量调整为与指定张量相同的大小。这相当于：

```
self.resize_(tensor.size())
```

## round()

参考 `torch.round()`

## round\_()

`round()` 的直接运算形式，即直接执行并且返回修改后的张量

## rsqrt()

参考 `torch.rsqrt()`

## rsqrt\_()

`rsqrt()` 的直接运算形式，即直接执行并且返回修改后的张量

## scatter\_(dim, index, src)

将 `src` 中的所有值按照 `index` 确定的索引写入本**tensor**中。其中索引是根据给定的**dimension**，`dim`按照 `gather()` 描述的规则来确定。

请注意，对于`collect`，`index`的值必须在0和`(self.size(dim)-1)`之间，包括所有值，并且指定维中的一行中的所有值必须是唯一的。

参数：

- `input (Tensor)`-源tensor
- `dim (int)`-索引的轴向
- `index (LongTensor)`-散射元素的索引指数
- `src (Tensor or float)`-散射的源元素

例子：

```
>>> x = torch.rand(2, 5)
>>> x

0.4319 0.6500 0.4080 0.8760 0.2355
0.2609 0.4711 0.8486 0.8573 0.1029
[torch.FloatTensor of size 2x5]

>>> torch.zeros(3, 5).scatter_(0, torch.LongTensor([[0, 1, 2, 0,
0], [2, 0, 0, 1, 2]]), x)

0.4319 0.4711 0.8486 0.8760 0.2355
0.0000 0.6500 0.0000 0.8573 0.0000
0.2609 0.0000 0.4080 0.0000 0.1029
[torch.FloatTensor of size 3x5]

>>> z = torch.zeros(2, 4).scatter_(1, torch.LongTensor([[2], [3]
]), 1.23)
>>> z

0.0000 0.0000 1.2300 0.0000
0.0000 0.0000 0.0000 1.2300
[torch.FloatTensor of size 2x4]
```

## select(dim, index) or number

按照`index`中选定的维度将`tensor`切片。如果`tensor`是一维的，则返回一个数字。否则，返回给定维度已经被移除的`tensor`。

参数：

- `dim (int)`-切片的维度
- `index (int)`-用来选取的索引

注意：`select()` 等效于切片。例如：`tensor.select(0, index)` 等效于 `tensor[index]`，`tensor.select(2, index)` 等效于 `tensor[:, :, index]`

## **set(source=None, storage\_offset=0, size=None, stride=None)**

设置底层内存，大小和步长。如果source是张量，则该张量将共享相同的存储空间，并且具有与给定张量相同的大小和步长。另一个则反映在一个张量内的元素变化。

参数：

- source (Tensor or Storage)-用到的tensor或内存
- storage\_offset (int)-内存的偏移量
- size (torch.Size)-需要的大小，默认为源tensor的大小。
- stride(tuple)-需要的步长，默认为C连续的步长。

## **sharememory()**

将底层存储器移动到共享内存。如果底层存储已经在共享内存和CUDA张量中，不进行任何操作。共享内存中的Tensors无法调整大小。

## **short()**

将tensor投射为short类型。

## **sigmoid()**

参考 `torch.sigmoid()`

## **sigmoid\_()**

`sidmoid()` 的直接运算形式，即直接执行并且返回修改后的张量

## **sign()**

参考 `torch.sign()`

## **sign\_()**

`sign()` 的直接运算形式，即直接执行并且返回修改后的张量

## **sin()**

参考 `torch.sin()`

## **sin\_()**

`sin()` 的直接运算形式，即直接执行并且返回修改后的张量

## **sinh()**

参考 `torch.sinh()`

## **sinh\_()**

`sinh()` 的直接运算形式，即直接执行并且返回修改后的张量

## **size() → torch.Size**

返回tensor的大小。返回的值是 `tuple` 的子类。

例：

```
>>> torch.Tensor(3, 4, 5).size()
torch.Size([3, 4, 5])
```

## **sort(dim=None, descending=False) -> (Tensor, LongTensor)**

参考 `torch.sort()`

## **split(split\_size, dim=0)**

将tensor分割成tensor数组。参考 `torch.split()`

## **sqrt()**

参考 `torch.sqrt()`

## **sqrt\_()**

`sqrt()` 的直接运算形式，即直接执行并且返回修改后的张量

## **squeeze(dim=None)**

参考 `torch.squeeze()`

## **squeeze\_(dim=None)**

`squeeze()` 的直接运算形式，即直接执行并且返回修改后的张量

**std() → float**

参考 `torch.std()`

**storage() → torch.Storage**

返回底层内存。

**storage\_offset() → int**

以储存元素的个数的形式返回tensor在地城内存中的偏移量。例：

```
>>> x = torch.Tensor([1, 2, 3, 4, 5])
>>> x.storage_offset()
0
>>> x[3:].storage_offset()
3
```

**classmethod() storage\_type()****stride()**

返回tesnor的步长。

**sub(value, other)**

从该张量中排除一个标量或张量。如果 `value` 和 `other` 都是给定的，则在使用之前 `other` 的每一个元素都会被 `value` 缩放。当 `other` 是一个张量，`other` 的形状必须可以与下面的张量的形状一起广播。

**sub\_(x)**

`sub()` 的直接运算形式，即直接执行并且返回修改后的张量

**sum(dim=None)**

参考 `torch.sum()`

**svd(some=True) -> (Tensor, Tensor, Tensor)**

参考 `torch.svd()`

**`symeig(eigenvectors=False, upper=True) -> (Tensor, Tensor)`**

参考 `torch.symeig()`

**`t()`**

参考 `torch.t()`

**`t()`**

`t()` 的直接运算形式，即直接执行并且返回修改后的张量

**`tan()`**

参考 `torch.tan()`

**`tan_()`**

`tan()` 的直接运算形式，即直接执行并且返回修改后的张量

**`tanh()`**

参考 `torch.tanh()`

**`tanh_()`**

`tanh()` 的直接运算形式，即直接执行并且返回修改后的张量

**`tolist()`**

返回一个tensor的嵌套列表表示。

**`topk(k, dim=None, largest=True, sorted=True) -> (Tensor, LongTensor)`**

参考 `torch.topk()`

**`trace() → float`**

参考 `torch.trace()`

## **transpose(dim0, dim1)**

参考 `torch.transpose()`

## **transpose(dim0, dim1)**

`transpose()` 的直接运算形式，即直接执行并且返回修改后的张量

## **tril(k=0)**

参考 `torch.tril()`

## **tril\_(k=0)**

`tril()` 的直接运算形式，即直接执行并且返回修改后的张量

## **triu(k=0)**

参考 `torch.triu()`

## **triu(k=0)**

`triu()` 的直接运算形式，即直接执行并且返回修改后的张量

## **trtrs(A, upper=True, transpose=False, unitriangular=False) -> (Tensor, Tensor)**

参考 `torch.trtrs()`

## **trunc()**

参考 `torch.trunc()`

## **trunc()**

`trunc()` 的直接运算形式，即直接执行并且返回修改后的张量

## **type(new\_type=None, async=False)**

如果未提供`new_type`，则返回类型，否则将此对象转换为指定的类型。如果已经是正确的类型，则不会执行且返回原对象。

参数：

- `new_type` (type或string) -需要的类型
- `async` (bool)-如果为True，并且源处于固定内存中，目标位于GPU上，反之亦然，则相对于主机异步执行该副本。否则，该参数不发挥作用。

## `type_as(tensor)`

将此张量转换为给定类型的张量。如果张量已经是正确的类型，则不会执行操作。等效于：

```
self.type(tensor.type())
```

参数：

- `tensor (Tensor)`:有所需要类型的tensor

## `unfold(dim, size, step)`

返回一个张量，其中包含尺寸`size`中所有尺寸的维度。如果`sizedim`是`dim`维度的原始大小，则返回张量中的维度是 $(sizedim-size)/step+1$

维度大小的附加维度将附加在返回的张量中。

参数：

- `dim (int)`- 展开的维度
- `size (int)`- 展开的每个切片的大小
- `step (int)`-相邻切片之间的步长

例子：



```
>>> x = torch.arange(1, 8)
>>> x

1
2
3
4
5
6
7
[torch.FloatTensor of size 7]

>>> x.unfold(0, 2, 1)

1 2
2 3
3 4
4 5
5 6
6 7
[torch.FloatTensor of size 6x2]

>>> x.unfold(0, 2, 2)

1 2
3 4
5 6
[torch.FloatTensor of size 3x2]
```

## uniform\_(from=0, to=1)

用均匀分布采样的数字填充该张量：

## unsqueeze(dim)

参考 `torch.unsqueeze()`

## unsqueeze\_(dim)

`unsqueeze()` 的直接运算形式，即直接执行并且返回修改后的张量

## var()

参考 `torch.var()`

## view(\*args)

返回具有相同数据但大小不同的新张量。返回的张量必须有与原张量相同的数据和相同数量的元素，但可以有不同的大小。一个张量必须是连续 `contiguous()` 的才能被查看。

例子：

```
>>> x = torch.randn(4, 4)
>>> x.size()
torch.Size([4, 4])
>>> y = x.view(16)
>>> y.size()
torch.Size([16])
>>> z = x.view(-1, 8) # the size -1 is inferred from other dimensions
>>> z.size()
torch.Size([2, 8])
```

## view\_as(tensor)

返回被视作与给定的tensor相同大小的原tensor。等效于：

```
self.view(tensor.size())
```

## zero\_()

用0填充这个张量。

## 译者署名

用户名	头像	职能	签名
Song		翻译	人生总要追求点什么

## Tensor Attributes

- `torch.dtype`
- `torch.device`
- `torch.layout`

每个 `torch.Tensor` 都有 `torch.dtype` , `torch.device` ,和 `torch.layout` 。

### `torch.dtype`

`torch.dtype` 是表示 `torch.Tensor` 的数据类型的对象。PyTorch 有八种不同的数据类型：

	Data type	dtype	Tensor types
32-bit floating point	<code>torch.float32</code> or <code>torch.float</code>	<code>torch.*.FloatTensor</code>	
64-bit floating point	<code>torch.float64</code> or <code>torch.double</code>	<code>torch.*.DoubleTensor</code>	
16-bit floating point	<code>torch.float16</code> or <code>torch.half</code>	<code>torch.*.HalfTensor</code>	
8-bit integer (unsigned)	<code>torch.uint8</code>	<code>torch.*.ByteTensor</code>	
8-bit integer (signed)	<code>torch.int8</code>	<code>torch.*.CharTensor</code>	
16-bit integer (signed)	<code>torch.int16</code> or <code>torch.short</code>	<code>torch.*.ShortTensor</code>	
32-bit integer (signed)	<code>torch.int32</code> or <code>torch.int</code>	<code>torch.*.IntTensor</code>	
64-bit integer (signed)	<code>torch.int64</code> or <code>torch.long</code>	<code>torch.*.LongTensor</code>	

使用方法：

```
>>> x = torch.Tensor([[1, 2, 3, 4, 5], [6, 7, 8, 9, 10]])
>>> print x.type()
torch.FloatTensor
```

### `torch.device`

`torch.device` 代表将 `torch.Tensor` 分配到的设备的对象。

`torch.device` 包含一个设备类型（`'cpu'` 或 `'cuda'` 设备类型）和可选的设备序号。如果设备序号不存在，则为当前设备；例如，`torch.Tensor` 用设备构建 `'cuda'` 的结果等同于 `'cuda:X'`，其中 `X` 是 `torch.cuda.current_device()` 的结果。

`torch.Tensor` 的设备可以通过 `Tensor.device` 访问属性。

构造 `torch.device` 可以通过字符串/字符串和设备编号。

通过一个字符串：

```
>>> torch.device('cuda:0')
device(type='cuda', index=0)

>>> torch.device('cpu')
device(type='cpu')

>>> torch.device('cuda') # current cuda device
device(type='cuda')
```

通过字符串和设备序号：

```
>>> torch.device('cuda', 0)
device(type='cuda', index=0)

>>> torch.device('cpu', 0)
device(type='cpu', index=0)
```

注意 `torch.device` 函数中的参数通常可以用一个字符串替代。这允许使用代码快速构建原型。

```
>>> # 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), 'cuda:1')
```

注意 出于传统原因，可以通过单个设备序号构建设备，将其视为 `cuda` 设备。这匹配 `Tensor.get_device()`，它为 `cuda` 张量返回一个序数，并且不支持 `cpu` 张量。

```
>>> torch.device(1)
device(type='cuda', index=1)
```

注意 指定设备的方法可以使用（properly formatted）字符串或（legacy）整数型设备序数，即以下示例均等效：

```
>>> torch.randn((2,3), device=torch.device('cuda:1'))
>>> torch.randn((2,3), device='cuda:1')
>>> torch.randn((2,3), device=1) # legacy
```

## torch.layout

`torch.layout` 表示 `torch.Tensor` 内存布局的对象。目前，我们支持 `torch.strided(dense Tensors)` 并为 `torch.sparse_coo(sparse COO Tensors)` 提供实验支持。

`torch.strided` 代表密集张量，是最常用的内存布局。每个 `strided` 张量都会关联一个 `torch.Storage`，它保存着它的数据。这些张量提供了多维度，存储的 `strided` 视图。`Strides` 是一个整数型列表：`k-th stride` 表示在张量的第 `k` 维从一个元素跳转到下一个元素所需的内存。这个概念使得可以有效地执行多张量。


例：

```
>>> x = torch.Tensor([[1, 2, 3, 4, 5], [6, 7, 8, 9, 10]])
>>> x.stride()
(5, 1)

>>> x.t().stride()
(1, 5)
```

关于 `torch.sparse_coo` 张量的更多信息,请参阅[torch.sparse](#)。

译者署名

用户名	头像	职能	签名
Song		翻译	人生总要追求点什么

## torch.sparse

警告 该API目前是实验性的，可能在不久的将来会发生变化。

torch支持COO（rdinate）格式的稀疏张量，可以有效地存储和处理大多数元素为零的张量。

稀疏张量被表示为一对致密张量：值的张量和2D张量的索引。可以通过提供这两个张量来构造稀疏张量，以及稀疏张量的大小（不能从这些张量推断出！）假设我们要在位置（0,2）处定义具有条目3的稀疏张量，位置（1,0）的条目4，位置（1,2）的条目5。我们会写：

```
>>> i = torch.LongTensor([[0, 1, 1],
 [2, 0, 2]])
>>> v = torch.FloatTensor([3, 4, 5])
>>> torch.sparse.FloatTensor(i, v, torch.Size([2,3])).to_dense()
 0 0 3
 4 0 5
[torch.FloatTensor of size 2x3]
```

请注意，LongTensor的输入不是索引元组的列表。如果要以这种方式编写索引，则在将它们传递给稀疏构造函数之前，应该进行转置：

```
>>> i = torch.LongTensor([[0, 2], [1, 0], [1, 2]])
>>> v = torch.FloatTensor([3, 4, 5])
>>> torch.sparse.FloatTensor(i.t(), v, torch.Size([2,3])).to_dense()
 0 0 3
 4 0 5
[torch.FloatTensor of size 2x3]
```

您还可以构建混合稀疏张量，其中只有第一个n维是稀疏的，其余的维度是密集的。

```
>>> i = torch.LongTensor([[2, 4]])
>>> v = torch.FloatTensor([[1, 3], [5, 7]])
>>> torch.sparse.FloatTensor(i, v).to_dense()
 0 0
 0 0
 1 3
 0 0
 5 7
[torch.FloatTensor of size 5x2]
```

可以通过指定一个空的稀疏张量来构建一个空的稀疏张量：

```
print torch.sparse.FloatTensor(2, 3)
FloatTensor of size 2x3 with indices:
[torch.LongTensor with no dimension]
and values:
[torch.FloatTensor with no dimension]
```

注意:

我们的稀疏张量格式允许未被缩小的稀疏张量，其中索引中可能有重复的坐标；在这种情况下，解释是该索引处的值是所有重复值条目的总和。非协调张量允许我们更有效地实施某些运营商。

在大多数情况下，您不必关心稀疏张量是否合并，因为大多数操作将在合并或未被缩小的稀疏张量的情况下工作相同。但是，您可能需要关心两种情况。

首先，如果您反复执行可以产生重复条目（例如 `torch.sparse.FloatTensor.add()`）的操作，则应偶尔将您的稀疏张量合并，以防止它们变得太大。

其次，一些运营商将取决于它们是否被合并或不产生不同的值（例如，`torch.sparse.FloatTensor._values()`和 `torch.sparse.FloatTensor._indices()`，以及 `torch.Tensor._sparse_mask()`）。这些运算符前面加上一个下划线，表示它们显示内部实现细节，应谨慎使用，因为与聚结的稀疏张量一起工作的代码可能无法与未被缩放的稀疏张量一起使用；一般来说，在与这些运营商合作之前明确地合并是最安全的。

例如，假设我们想通过直接操作来实现一个操作符 `torch.sparse.FloatTensor._values()`。随着乘法分布的增加，标量的乘法可以以明显的方式实现；然而，平方根不能直接实现，因为（如果你被赋予了未被缩放的张量，那将是什么）。 $\text{sqrt}(a + b) \neq \text{sqrt}(a) + \text{sqrt}(b)$

## class torch.sparse.FloatTensor

- `add()`
- `add_()`
- `clone()`
- `dim()`
- `div()`
- `div_()`
- `get_device()`
- `hspmm()`
- `mm()`
- `mul()`
- `mul_()`
- `resizeAs_()`
- `size()`
- `spadd()`
- `spmm()`



- `sspaddmm()`
- `sspmm()`
- `sub()`
- `sub_()`
- `t_()`
- `toDense()`
- `transpose()`
- `transpose_()`
- `zero_()`
- `coalesce()`
- `is_coalesced()`
- `_indices()`
- `_values()`
- `_nnz()`

译者署名

用户名	头像	职能	签名
Song		翻译	人生总要追求点什么

## torch.cuda

---

- [交流集](#)
- [流和事件](#)
- [NVIDIA工具扩展 \(NVTX\)](#)

---

该包增加了对CUDA张量类型的支持，实现了与CPU张量相同的功能，但使用GPU进行计算。

它是延迟的初始化，所以你可以随时导入它，并使用 `is_available()` 来确定系统是否支持CUDA。

[CUDA语义](#) 有关于使用CUDA的更多细节。

```
torch.cuda.current_blas_handle()
```

返回指向当前cuBLAS句柄的cublasHandle\_t指针

```
torch.cuda.current_device()
```

返回当前所选设备的索引。

```
torch.cuda.current_stream()
```

返回当前选定的 `Stream`

```
class torch.cuda.device(idx)
```

更改所选设备的上下文管理器。

参数：

- `idx(int)` – 设备索引选择。如果这个参数是负的，则是无效操作。

```
torch.cuda.device_count()
```

返回可用的GPU数量。

```
class torch.cuda.device_of(obj)
```

将当前设备更改为给定对象的上下文管理器。

可以使用张量和存储作为参数。如果给定的对象不是在GPU上分配的，则是无效操作。

参数：

- **obj (Tensor或者Storage)** – 在选定设备上分配的对象。

```
torch.cuda.is_available()
```

返回bool值，指示当前CUDA是否可用。

```
torch.cuda.set_device(device)
```

设置当前设备。

不鼓励使用此功能函数。在大多数情况下，最好使用 `CUDA_VISIBLE_DEVICES` 环境变量。

参数：

- **device(int)** - 选择的设备。如果此参数为负，则此函数是无操作的。

```
torch.cuda.stream(stream)
```

选择给定流的上下文管理器。

在其上下文中排队的所有CUDA核心将在所选流上排列。

参数：

- **stream(Stream)** – 选择的流。如果为 `None`，则这个管理器是无效的。

```
torch.cuda.synchronize()
```

等待当前设备上所有流中的所有内核完成。

## 交流集

```
torch.cuda.comm.broadcast(tensor, devices)
```

向一些GPU广播张量。

参数：

- `tensor (Tensor)` – 广播的张量
- `devices (Iterable)` – 可以广播的设备的迭代。注意，它的设备形式为 (`src`, `dst1`, `dst2`, ...)，其第一个元素是广播来源的设备。

返回：包含张量副本的元组，放置在对应于索引的设备上。

```
torch.cuda.comm.reduce_add(inputs, destination=None)
```

将来自多个GPU的张量相加。

所有输入应具有匹配的形狀。

参数：

- `inputs (Iterable[Tensor])` – 要相加张量的迭代
- `destination (int, 可选)` – 将放置输出的设备（默认值：当前设备）。

返回：包含放置在 `destination` 设备上的所有输入的元素总和的张量。

```
torch.cuda.comm.scatter(tensor, devices, chunk_sizes=None, dim=0, streams=None)
```

跨多个GPU分散张量。

参数：

- `tensor (Tensor)` – 要分散的张量
- `devices (Iterable[int])` – 可选代的int，指定张量应分散在哪些设备中。
- `chunk_sizes (Iterable[int], 可选)` – 要放置在每个设备上的块大小。它应该匹配 `devices` 的长度且总和为 `tensor.size(dim)`。如果没有指定，张量将被分成相等的块。
- `dim (int, 可选)` – 将张量块化的一个维度。

返回：包含 `tensor` 块的元组，传播给 `devices`。

```
torch.cuda.comm.gather(tensors, dim=0, destination=None)
```

从多个GPU收集张量。

所有张量的测量尺寸与`dim`不一致。

参数：

- `tensors (Iterable[Tensor])` – 要收集的张量的迭代。
- `dim (int)` – 张量沿着此维度来连接。

- **destination** (int, 可选) – 输出设备（-1表示CPU，默认值：当前设备）。

返回：一个张量位于 **destination** 设备上，这是沿着 **dim** 连接 **tensors** 的结果。

## 流和事件

```
class torch.cuda.Stream
```

CUDA流的包装。

参数：

- **device** (int, 可选) – 分配流的设备。
- **priority** (int, 可选) – 流的优先级。数字越小优先级越高。

> **query()**

检查所有提交的工作是否已经完成。

返回：一个布尔值，表示此流中的所有核心是否完成。

> **record\_event(event=None)**

记录事件。

参数：

- **event** (Event, 可选) – 要记录的事件。如果未定义，将分配一个新的。返回：记录的事件。

> **synchronize()**

等待此流中的所有核心完成。

> **wait\_event(event)**

将所有未来的工作提交到流等待事件。

参数：**event** (Event) – 等待的事件

> **wait\_stream(stream)**

与另一个流同步。

提交到此流的所有未来工作将等待直到所有核心在调用完成时提交给给定的流。

```
class torch.cuda.Event(enable_timing=False, blocking=False, interrupt_process=False, _handle=None)
```

CUDA事件的包装。

参数：

- `enable_timing (bool)` – 指示事件是否应该测量时间（默认值：`False`）
- `blocking (bool)` – 如果为`true`，`wait()` 将被阻塞（默认值：`False`）
- `interprocess (bool)` – 如果为`true`，事件可以在进程之间共享（默认值：`False`）

> `elapsed_time(end_event)`

返回事件记录之前经过的时间。

> `ipc_handle()`

返回此事件的IPC句柄。

> `query()`

检查事件是否已被记录。

返回：指示事件是否已被记录的布尔值。

> `record(stream=None)`

在给定的流中记录事件。

> `synchronize()`

与事件同步。

> `wait(stream=None)`

使给定的流等待事件。

## NVIDIA工具扩展（NVTX）

```
torch.cuda.nvtx.mark(msg)
```

描述在某个时刻发生的瞬时事件。参数：

- `msg (string)` - 与事件关联的ASCII消息。

```
torch.cuda.nvtx.range_push(msg)
```

将范围推送到嵌套范围跨度的堆栈。返回起始范围的基于zero-based 的深度。  
参数：

- `msg (string)` - 与范围关联的ASCII消息

```
torch.cuda.nvtx.range_pop()
```

从一堆嵌套的范围范围内弹出一个范围。返回结束的范围的基于zero-based的深度。

译者署名

用户名	头像	职能	签名
Song		翻译	人生总要追求点什么

# torch.Storage

`torch.Storage` 是单个数据类型的连续的 一维数组 ，每个 `torch.Tensor` 都具有相同数据类型的相应存储。

```
class torch.FloatTensor
```

- `byte()` : 将Storage转为byte类型
- `char()` : 将Storage转为char类型
- `clone()` : 返回Storage的副本
- `copy_()`
- `cpu()` : 如果尚未在CPU上，则返回Storage的CPU副本
- `cuda(device=None, async=False)` : 在CUDA内存中返回此对象的副本。如果该对象已经在CUDA内存中，并且在正确的设备上，则不会执行任何操作，并返回原始对象。 参数说明：
  1. `device (int)` - 目标GPU的id。默认值是当前设备。
  2. `async (bool)` - 如果值为True，且源在锁定内存中，则副本相对于宿主是异步的。否则此参数不起效果。
- `data_ptr()` : 返回一个时间戳
- `double()` : 将Storage转为double类型
- `element_size()` : 返回参数的size
- `fill_()`
- `float()` : 将Storage转为float类型
- `from_buffer()`
- `half()` : 将Storage转为half类型
- `int()` : 将Storage转为int类型
- `is_cuda = False`
- `is_pinned()`
- `is_shared()`
- `is_sparse = False`
- `long()` : 将Storage转为long类型
- `new()`
- `pin_memory()` : 将存储复制到固定内存（如果尚未固定）。
- `resize_()`
- `share_memory_()` : 这对于已经在共享内存和CUDA存储器中的存储器是无效的，不需要为了跨进程共享而移动。无法调整共享内存中的存储空间。返回：`self`
- `short()` : 将Storage转为short类型
- `size()` : 返回Storage转的大小
- `tolist()` : 返回一个包含Storage中元素的列表



- `type(new_type=None, async=False)`：将此对象转为指定类型。如果已经是正确类型，不会执行复制操作，直接返回原对象。

参数说明：

1. `new_type` (`type`或`string`) -需要转成的类型
2. `async` (`bool`) -如果值为`True`，并且源处于固定内存中，目标位于GPU上，反之亦然，则相对于主机异步执行该副本。否则，参数没有效果。

具体使用教程请参考：[使用torch.Storage共享多个张量的相同存储以及将torch.Tensor转化为torch.Storage](#)

译者署名

用户名	头像	职能	签名
Song		翻译	人生总要追求点什么

## torch.nn

---

- [Parameters](#)
- [Containers](#)

## Parameters

### class torch.nn.Parameter()

一种 `Variable`，被视为一个模块参数。

`Parameters` 是 `Variable` 的子类。当与 `Module` 一起使用时，它们具有非常特殊的属性，当它们被分配为模块属性时，它们被自动添加到其参数列表中，并将出现在例如 `parameters()` 迭代器中。分配变量没有这样的效果。这是因为人们可能希望在模型中缓存一些临时状态，如 `RNN` 的最后一个隐藏状态。如果没有这样的班级 `Parameter`，这些临时人员也会注册。

另一个区别是，`parameters` 不能是 `volatile`，他们默认要求梯度。

参数说明:

- `data (Tensor)` – parameter tensor.
- `requires_grad (bool, optional)` – 如果需要计算剃度，可以参考[从向后排除子图](#)

## Containers：

### class torch.nn.Module

所有神经网络模块的基类。

你的模型也应该继承这个类。

`Modules` 还可以包含其他模块，允许将它们嵌套在树结构中。您可以将子模块分配为常规属性：

```
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) # submodule: Conv2d
 self.conv2 = nn.Conv2d(20, 20, 5)

 def forward(self, x):
 x = F.relu(self.conv1(x))
 return F.relu(self.conv2(x))
```

以这种方式分配的子模块将被注册，并且在调用 `.cuda()` 等时也会转换参数。

## add\_module(name, module)

将一个子模块添加到当前模块。该模块可以使用给定的名称作为属性访问。例：

```
import torch.nn as nn
class Model(nn.Module):
 def __init__(self):
 super(Model, self).__init__()
 self.add_module("conv", nn.Conv2d(10, 20, 4))
 #self.conv = nn.Conv2d(10, 20, 4) 和上面这个增加module的方
式等价
model = Model()
print(model.conv)
```

输出：

```
Conv2d(10, 20, kernel_size=(4, 4), stride=(1, 1))
```

## apply(fn)

适用 `fn` 递归到每个子模块（如返回 `.children()`），以及自我。典型用途包括初始化模型的参数（另见 `torch.nn.init`）。例如：

```

>>> 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 (2 -> 2)
Parameter containing:
 1 1
 1 1
[torch.FloatTensor of size 2x2]
Linear (2 -> 2)
Parameter containing:
 1 1
 1 1
[torch.FloatTensor of size 2x2]
Sequential (
 (0): Linear (2 -> 2)
 (1): Linear (2 -> 2)
)

```

## children()

返回直接的子模块的迭代器。

## cpu(device\_id=None)

将所有模型参数和缓冲区移动到CPU

## cuda(device\_id=None)

将所有模型参数和缓冲区移动到GPU。

参数说明:

- `device_id` (int, 可选) – 如果指定，所有参数将被复制到该设备

## double()

将所有参数和缓冲区转换为双数据类型。

## eval()

将模型设置成 `evaluation` 模式

仅仅当模型中有 `Dropout` 和 `BatchNorm` 是才会有影响。

## **float()**

将所有参数和缓冲区转换为`float`数据类型。

## **forward(\* input)**

定义计算在每一个调用执行。应该被所有子类重写。

## **half()**

将所有参数和缓冲区转换为 `half` 类型。

## **load\_state\_dict(state\_dict)**

将参数和缓冲区复制 `state_dict` 到此模块及其后代。键 `state_dict` 必须与此模块 `state_dict()` 功能返回的键完全相符。

参数说明:

- `state_dict (dict)` – 保存 `parameters` 和 `persistent buffers` 的 `dict` 。

## **modules()**

返回网络中所有模块的迭代器。

**NOTE：** 重复的模块只返回一次。在以下示例中，`1` 将仅返回一次。

```
>>> l = nn.Linear(2, 2)
>>> net = nn.Sequential(l, l)
>>> for idx, m in enumerate(net.modules()):
>>> print(idx, '->', m)
0 -> Sequential (
 (0): Linear (2 -> 2)
 (1): Linear (2 -> 2)
)
1 -> Linear (2 -> 2)
```

## **named\_children()**

返回包含子模块的迭代器，同时产生模块的名称以及模块本身。

例子：

```
>>> for name, module in model.named_children():
>>> if name in ['conv4', 'conv5']:
>>> print(module)
```

## named\_modules(memo=None, prefix="")

返回网络中所有模块的迭代器，同时产生模块的名称以及模块本身。

注意：重复的模块只返回一次。在以下示例中，`l` 将仅返回一次。

```
>>> l = nn.Linear(2, 2)
>>> net = nn.Sequential(l, l)
>>> for idx, m in enumerate(net.named_modules()):
>>> print(idx, '->', m)
0 -> ('', Sequential (
 (0): Linear (2 -> 2)
 (1): Linear (2 -> 2)
))
1 -> ('0', Linear (2 -> 2))
```

## named\_parameters(memo=None, prefix="")

返回模块参数的迭代器，同时产生参数的名称以及参数本身 例如：

```
>>> for name, param in self.named_parameters():
>>> if name in ['bias']:
>>> print(param.size())
```

## parameters()

返回模块参数的迭代器。这通常被传递给优化器。

例子：

```
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)

在模块上注册一个向后的钩子。

每当计算相对于模块输入的梯度时，将调用该钩。挂钩应具有以下签名：

```
hook(module, grad_input, grad_output) -> Variable or None
```

如果 `module` 有多个输入输出的话，那么 `grad_input` `grad_output` 将会是个 `tuple`。 `hook` 不应该修改它的 `arguments`，但是它可以选择性的返回关于输入的梯度，这个返回的梯度在后续的计算中会替代 `grad_input`。

这个函数返回一个句柄( `handle` )。它有一个方法 `handle.remove()`，可以用这个方法将 `hook` 从 `module` 移除。

## register\_buffer(name, tensor)

给 `module` 添加一个持久缓冲区。

这通常用于注册不应被视为模型参数的缓冲区。例如，`BatchNorm running_mean` 不是参数，而是持久状态的一部分。

缓冲区可以使用给定的名称作为属性访问。

例子：

```
self.register_buffer('running_mean', torch.zeros(num_features))
```

## register\_forward\_hook(hook)

在模块上注册一个 `forward hook`。每次调用 `forward()` 计算输出的时候，这个 `hook` 就会被调用。它应该拥有以下签名：

```
hook(module, input, output) -> None
```

`hook` 不应该修改 `input` 和 `output` 的值。这个函数返回一个有 `handle.remove()` 方法的句柄( `handle` )。可以用这个方法将 `hook` 从 `module` 移除。

## register\_parameter(name, param)

向 `module` 添加 `parameter`

该参数可以使用给定的名称作为属性访问。

## state\_dict(destination=None, prefix="")

返回包含模块整体状态的字典。

包括参数和持久缓冲区（例如运行平均值）。键是相应的参数和缓冲区名称。

例子：

```
module.state_dict().keys()
['bias', 'weight']
```

## train(mode=True)

将模块设置为训练模式。

仅仅当模型中有 `Dropout` 和 `BatchNorm` 是才会有影响。

## zero\_grad()

将所有模型参数的梯度设置为零。

## class torch.nn.Sequential(\* args)

一个时序容器。`Modules` 会以他们传入的顺序被添加到容器中。当然，也可以传入一个 `OrderedDict`。

为了更容易理解，给出的是一个小例子：

```
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())
]))
```

## class torch.nn.ModuleList(modules=None)

将 `submodules` 保存在一个 `list` 中。

`ModuleList` 可以像一般的 `Python list` 一样被索引。而且 `ModuleList` 中包含的 `modules` 已经被正确的注册，对所有的 `module method` 可见。



参数说明:

- `modules (list, optional)` – 要添加的模块列表

例子:

```
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)

在列表末尾附加一个给定的模块。

参数说明:

- `module (nn.Module)` – 要追加的模块

## extend(modules)

最后从Python列表中追加模块。

参数说明:

- `modules(list)` – 要附加的模块列表

## class torch.nn.ParameterList(parameters=None)

在列表中保存参数。

`ParameterList`可以像普通Python列表一样进行索引，但是它包含的参数已经被正确注册，并且将被所有的Module方法都可见。

参数说明:

- `modules (list, 可选)` – `nn.Parameter`要添加的列表

例子:

```
class MyModule(nn.Module):
 def __init__(self):
 super(MyModule, self).__init__()
 self.params = nn.ParameterList([nn.Parameter(torch.randn(
10, 10)) for i in range(10)])

 def forward(self, x):
 # ModuleList can act as an iterable, or be indexed using
ints
 for i, p in enumerate(self.params):
 x = self.params[i // 2].mm(x) + p.mm(x)
 return x
```

## append(parameter)

在列表末尾添加一个给定的参数。

参数说明:

- parameter (nn.Parameter) – 要追加的参数

## extend(parameters)

在Python列表中附加参数。

参数说明:

- parameters (list) – 要追加的参数列表

## 卷积层

**class torch.nn.Conv1d(in\_channels, out\_channels, kernel\_size, stride=1, padding=0, dilation=1, groups=1, bias=True)**

一维卷积层，输入的尺度是(N, C<sub>in</sub>, L)，输出尺度 (N, C<sub>out</sub>, L<sub>out</sub>) 的计算方式：

$$out(N_i, C_{out_j}) = bias(C_{out_j}) + \sum_{k=0}^{C_{in}-1} weight(C_{out_j}, k) \cdot input(N_i, k)$$

说明

**bigotimes** : 表示相关系数计算 **stride** : 控制相关系数的计算步长  
**dilation** : 用于控制内核点之间的距离，详细描述在[这里](#) **groups** : 控制输入和输出之间的连接， **group=1** ，输出是所有的输入的卷积； **group=2** ，此时相当

于有并排的两个卷积层，每个卷积层计算输入通道的一半，并且产生的输出是输出通道的一半，随后将这两个输出连接起来。

### Parameters :

- `in_channels( int )` – 输入信号的通道
- `out_channels( int )` – 卷积产生的通道
- `kerner_size( int or tuple )` - 卷积核的尺寸
- `stride( int or tuple , optional )` - 卷积步长
- `padding( int or tuple , optional )` - 输入的每一条边补充0的层数
- `dilation( int or tuple , `optional` )` – 卷积核元素之间的间距
- `groups( int , optional )` – 从输入通道到输出通道的阻塞连接数
- `bias( bool , optional )` - 如果 `bias=True` ，添加偏置

**shape:** 输入: (N,C<sub>in</sub>,L<sub>in</sub>) 输出: (N,C<sub>out</sub>,L<sub>out</sub>) 输入输出的计算方式：

$$L_{out} = \text{floor}((L_{in} + 2padding - dilation(kerner\_size - 1) - 1) / stride + 1)$$

变量：

- `weight( tensor )` - 卷积的权重，大小是( `out_channels` , `in_channels` , `kernel_size` )
- `bias( tensor )` - 卷积的偏置系数，大小是 ( `out_channel` )

### example:

```
>>> m = nn.Conv1d(16, 33, 3, stride=2)
>>> input = autograd.Variable(torch.randn(20, 16, 50))
>>> output = m(input)
```

**class torch.nn.Conv2d(in\_channels, out\_channels, kernel\_size, stride=1, padding=0, dilation=1, groups=1, bias=True)**

二维卷积层，输入的尺度是(N, C<sub>in</sub>,H,W)，输出尺度 (N,C<sub>out</sub>,H<sub>out</sub>,W<sub>out</sub>) 的计算方式：

$$out(N_i, C_{outj}) = bias(C_{outj}) + \sum^{C_{in}-1}_{k=0} weight(C_{out\_j}, k) \bigotimes input(N_i, k)$$

说明 `bigotimes`：表示二维的相关系数计算 `stride`：控制相关系数的计算步长  
`dilation`：用于控制内核点之间的距离，详细描述在[这里](#) `groups`：控制输入和输出之间的连接：  
`group=1`，输出是所有的输入的卷积；`group=2`，此时相当于有并排的两个卷积层，每个卷积层计算输入通道的一半，并且产生的输出是输出通道的一半，随后将这两个输出连接起来。

参数 `kernel_size` , `stride`,`padding` , `dilation` 也可以是一个 `int` 的数据, 此时卷积`height`和`width`值相同;也可以是一个 `tuple` 数组, `tuple` 的第一维度表示`height`的数值, `tuple`的第二维度表示`width`的数值

### Parameters :

- `in_channels( int )` - 输入信号的通道
- `out_channels( int )` - 卷积产生的通道
- `kerner_size( int or tuple )` - 卷积核的尺寸
- `stride( int or tuple , optional )` - 卷积步长
- `padding( int or tuple , optional )` - 输入的每一条边补充0的层数
- `dilation( int or tuple , optional )` - 卷积核元素之间的间距
- `groups( int , optional )` - 从输入通道到输出通道的阻塞连接数
- `bias( bool , optional )` - 如果 `bias=True` , 添加偏置

**shape:** input: (N,C\_in,H\_in,W\_in) output: (N,C\_out,H\_out,W\_out)

$$H_{out} = \text{floor}((H_{in} + 2padding[0] - dilation[0](kernel\_size[0] - 1) - 1) / stride[0] + 1)$$

$$W_{out} = \text{floor}((W_{in} + 2padding[1] - dilation[1](kernel\_size[1] - 1) - 1) / stride[1] + 1)$$

变量: `weight( tensor )` - 卷积的权重, 大小是( `out_channels` , `in_channels` , `kernel_size` ) `bias( tensor )` - 卷积的偏置系数, 大小是( `out_channel` )

### Examples:

```
>>> # With square kernels and equal stride
>>> m = nn.Conv2d(16, 33, 3, stride=2)
>>> # non-square kernels and unequal stride and with padding
>>> m = nn.Conv2d(16, 33, (3, 5), stride=(2, 1), padding=(4, 2))
>>> # non-square kernels and unequal stride and with padding and
 dilation
>>> m = nn.Conv2d(16, 33, (3, 5), stride=(2, 1), padding=(4, 2),
 dilation=(3, 1))
>>> input = autograd.Variable(torch.randn(20, 16, 50, 100))
>>> output = m(input)
```

**class torch.nn.Conv3d(in\_channels, out\_channels, kernel\_size, stride=1, padding=0, dilation=1, groups=1, bias=True)**

三维卷积层, 输入的尺度是(N, C\_in,D,H,W), 输出尺度(N,C\_out,D\_out,H\_out,W\_out)的计算方式:

$$out(N_i, C_{out_j}) = bias(C_{out_j}) + \sum_{k=0}^{C_{in}-1} weight(C_{out\_j}, k) \otimes input(N_i, k)$$

说明 `bigotimes` :表示二维的相关系数计算 `stride` :控制相关系数的计算步长  
`dilation` :用于控制内核点之间的距离, 详细描述在[这里](#) `groups` :控制输入和输出之间的连接: `group=1` , 输出是所有的输入的卷积; `group=2` , 此时相当于有并排的两个卷积层, 每个卷积层计算输入通道的一半, 并且产生的输出是输出通道的一半, 随后将这两个输出连接起来。参

数 `kernel_size` , `stride` , `padding` , `dilation` 可以是一个 `int` 的数据 - 卷积height和width值相同, 也可以是一个有三个 `int` 数据的 `tuple` 数组, `tuple` 的第一维度表示depth的数值, `tuple` 的第二维度表示height的数值, `tuple` 的第三维度表示width的数值

## Parameters :

- `in_channels( int )` - 输入信号的通道
- `out_channels( int )` - 卷积产生的通道
- `kernel_size( int or tuple )` - 卷积核的尺寸
- `stride( int or tuple , optional )` - 卷积步长
- `padding( int or tuple , optional )` - 输入的每一条边补充0的层数
- `dilation( int or tuple , optional )` - 卷积核元素之间的间距
- `groups( int , optional )` - 从输入通道到输出通道的阻塞连接数
- `bias( bool , optional )` - 如果 `bias=True` , 添加偏置

**shape:** `input : ((N, C{in}, D{in}, H{in}, W{in})) output : ((N, C{out}, D{out}, H{out}, W{out}))` where  $(D\{out\} = \text{floor}((D\{in\} + 2 \text{padding}[0] - dilation[0] (\text{kernel\_size}[0] - 1) - 1) / \text{stride}[0] + 1))$   $(H\{out\} = \text{floor}((H\{in\} + 2 \text{padding}[1] - dilation[1] (\text{kernel\_size}[1] - 1) - 1) / \text{stride}[1] + 1))$   $(W\{out\} = \text{floor}((W\{in\} + 2 \text{padding}[2] - dilation[2] (\text{kernel\_size}[2] - 1) - 1) / \text{stride}[2] + 1))$

变量:

- `weight( tensor )` - 卷积的权重, shape是 `( out_channels , in_channels , kernel_size )`
- `bias( tensor )` - 卷积的偏置系数, shape是 `( out_channel )`

Examples:

```
>>> # With square kernels and equal stride
>>> m = nn.Conv3d(16, 33, 3, stride=2)
>>> # non-square kernels and unequal stride and with padding
>>> m = nn.Conv3d(16, 33, (3, 5, 2), stride=(2, 1, 1), padding=(4, 2, 0))
>>> input = autograd.Variable(torch.randn(20, 16, 10, 50, 100))
>>> output = m(input)
```

**class torch.nn.ConvTranspose1d(in\_channels, out\_channels, kernel\_size, stride=1, padding=0, output\_padding=0, groups=1, bias=True, dilation=1)**

1维的解卷积操作（`transposed convolution operator`，注意改视作操作可视为解卷积操作，但并不是真正的解卷积操作）该模块可以看作是 `Conv1d` 相对于其输入的梯度，有时（但不正确地）被称为解卷积操作。

注意 由于内核的大小，输入的最后的一些列的数据可能会丢失。因为输入和输出不是完全的互相关。因此，用户可以进行适当的填充（padding操作）。

参数

- `in_channels( int )` - 输入信号的通道数
- `out_channels( int )` - 卷积产生的通道
- `kernel_size( int or tuple )` - 卷积核的大小
- `stride( int or tuple , optional )` - 卷积步长
- `padding( int or tuple , optional )` - 输入的每一条边补充0的层数
- `output_padding( int or tuple , optional )` - 输出的每一条边补充0的层数
- `dilation( int or tuple , optional )` - 卷积核元素之间的间距
- `groups( int , optional )` - 从输入通道到输出通道的阻塞连接数
- `bias( bool , optional )` - 如果 `bias=True`，添加偏置

**shape:** 输入:  $((N, C_{in}, L_{in}))$  输出:  $((N, C_{out}, L_{out}))$  where  $(L_{out} = (L_{in} - 1) \times stride - 2 \times padding + kernel\_size + output\_padding)$

变量:

- `weight( tensor )` - 卷积的权重，大小是  $(in\_channels, in\_channels, kernel\_size)$
- `bias( tensor )` - 卷积的偏置系数，大小是  $(out\_channel)$

**class torch.nn.ConvTranspose2d(in\_channels, out\_channels, kernel\_size, stride=1, padding=0, output\_padding=0, groups=1, bias=True, dilation=1)**

2维的转置卷积操作（`transposed convolution operator`，注意改视作操作可视为解卷积操作，但并不是真正的解卷积操作）该模块可以看作是 `Conv2d` 相对于其输入的梯度，有时（但不正确地）被称为解卷积操作。

说明

`stride`: 控制相关系数的计算步长 `dilation`: 用于控制内核点之间的距离，详细描述在[这里](#) `groups`: 控制输入和输出之间的连接: `group=1`，输出是所有的输入的卷积; `group=2`，此时相当于有并排的两个卷积层，每个卷积层计算输入通道的一半，并且产生的输出是输出通道的一半，随后将这两个输出连接起来。

参数 `kernel_size`，`stride`，`padding`，`dilation` 数据类型: 可以是一个 `int` 类型的数据，此时卷积height和width值相同;也可以是一个 `tuple` 数组（包含来两个 `int` 类型的数据），第一个 `int` 数据表示 `height` 的数值，第二个 `int` 类型的数据表示width的数值



注意 由于内核的大小，输入的最后的一些列的数据可能会丢失。因为输入和输出是不是完全的互相关。因此，用户可以进行适当的填充（padding 操作）。

参数：

- `in_channels( int )` – 输入信号的通道数
- `out_channels( int )` – 卷积产生的通道数
- `kerner_size( int or tuple )` - 卷积核的大小
- `stride( int or tuple , optional )` - 卷积步长
- `padding( int or tuple , optional )` - 输入的每一条边补充0的层数
- `output_padding( int or tuple , optional )` - 输出的每一条边补充0的层数
- `dilation( int or tuple , optional )` – 卷积核元素之间的间距
- `groups( int , optional )` – 从输入通道到输出通道的阻塞连接数
- `bias( bool , optional )` - 如果 `bias=True` ，添加偏置

**shape:** 输入:  $((N, C\{in\}, H\{in\}, W\{in\}))$  输出:  $((N, C\{out\}, H\{out\}, W\{out\}))$  where  
 $H\{out\} = (H\{in\} - 1) \text{ stride}[0] - 2 \text{ padding}[0] + \text{kernelsize}[0] + \text{output\_padding}[0]$   
 $W\{out\} = (W\{in\} - 1) \text{ _stride}[1] - 2 \text{ padding}[1] + \text{kernel\_size}[1] + \text{output\_padding}[1]$

变量：

- `weight( tensor )` - 卷积的权重，大小是  $(\text{in\_channels}, \text{in\_channels}, \text{kernel\_size})$
- `bias( tensor )` - 卷积的偏置系数，大小是  $(\text{out\_channel})$

## Example

```
>>> # With square kernels and equal stride
>>> m = nn.ConvTranspose2d(16, 33, 3, stride=2)
>>> # non-square kernels and unequal stride and with padding
>>> m = nn.ConvTranspose2d(16, 33, (3, 5), stride=(2, 1), padding=(4, 2))
>>> input = autograd.Variable(torch.randn(20, 16, 50, 100))
>>> output = m(input)
>>> # exact output size can be also specified as an argument
>>> input = autograd.Variable(torch.randn(1, 16, 12, 12))
>>> downsample = nn.Conv2d(16, 16, 3, stride=2, padding=1)
>>> upsample = nn.ConvTranspose2d(16, 16, 3, stride=2, padding=1)
>>> h = downsample(input)
>>> h.size()
torch.Size([1, 16, 6, 6])
>>> output = upsample(h, output_size=input.size())
>>> output.size()
torch.Size([1, 16, 12, 12])
```

```
class torch.nn.ConvTranspose3d(in_channels,
out_channels, kernel_size, stride=1, padding=0,
output_padding=0, groups=1, bias=True, dilation=1)
```

3维的转置卷积操作（`transposed convolution operator`，注意改视作操作可视为解卷积操作，但并不是真正的解卷积操作）转置卷积操作将每个输入值和一个可学习权重的卷积核相乘，输出所有输入通道的求和

该模块可以看作是 `Conv3d` 相对于其输入的梯度，有时（但不正确地）被称为解卷积操作。

说明

`stride`：控制相关系数的计算步长 `dilation`：用于控制内核点之间的距离，详细描述在[这里](#) `groups`：控制输入和输出之间的连接：`group=1`，输出是所有的输入的卷积；`group=2`，此时相当于有并排的两个卷积层，每个卷积层计算输入通道的一半，并且产生的输出是输出通道的一半，随后将这两个输出连接起来。

参数 `kernel_size`，`stride`，`padding`，`dilation` 数据类型：一个 `int` 类型的数据，此时卷积height和width值相同；也可以是一个 `tuple` 数组（包含来两个 `int` 类型的数据），第一个 `int` 数据表示height的数值，`tuple`的第二个`int`类型的数据表示width的数值

注意 由于内核的大小，输入的最后的一些列的数据可能会丢失。因为输入和输出是不是完全的互相关。因此，用户可以进行适当的填充（`padding`操作）。

参数：

- `in_channels( int )` – 输入信号的通道数
- `out_channels( int )` – 卷积产生的通道数
- `kernel_size( int or tuple )` - 卷积核的大小
- `stride( int or tuple , optional )` - 卷积步长
- `padding( int or tuple , optional )` - 输入的每一条边补充0的层数
- `output_padding( int or tuple , optional )` - 输出的每一条边补充0的层数
- `dilation( int or tuple , optional )` – 卷积核元素之间的间距
- `groups( int , optional )` – 从输入通道到输出通道的阻塞连接数
- `bias( bool , optional )` - 如果 `bias=True`，添加偏置

**shape:** 输入:  $((N, C_{in}, D_{in}, H_{in}, W_{in}))$  输出:  $((N, C_{out}, D_{out}, H_{out}, W_{out}))$  where  $(D_{out} = (D_{in} - 1) \text{stride}[0] - 2 \text{padding}[0] + \text{kernelsize}[0] + \text{output\_padding}[0])$   $(H_{out} = (H_{in} - 1) \text{stride}[1] - 2 \text{padding}[1] + \text{kernelsize}[1] + \text{output\_padding}[1])$   $(W_{out} = (W_{in} - 1) \text{stride}[2] - 2 \text{padding}[2] + \text{kernel\_size}[2] + \text{output\_padding}[2])$

变量：

- `weight( tensor )` - 卷积的权重，大小是  $(\text{in\_channels}, \text{in\_channels}, \text{kernel\_size})$



- `bias( tensor )` - 卷积的偏置系数，大小是 ( `out_channel` )

## Example

```
>>> # With square kernels and equal stride
>>> m = nn.ConvTranspose3d(16, 33, 3, stride=2)
>>> # non-square kernels and unequal stride and with padding
>>> m = nn.Conv3d(16, 33, (3, 5, 2), stride=(2, 1, 1), padding=(0, 4, 2))
>>> input = autograd.Variable(torch.randn(20, 16, 10, 50, 100))
>>> output = m(input)
```

## 池化层

**`class torch.nn.MaxPool1d(kernel_size, stride=None, padding=0, dilation=1, return_indices=False, ceil_mode=False)`**

对于输入信号的输入通道，提供1维最大池化 ( `max pooling` ) 操作

如果输入的大小是(N,C,L)，那么输出的大小是(N,C,L<sub>out</sub>)的计算方式是：

$$out(N_i, C_j, k) = \max^{kernel\_size-1}_{m=0} input(N_i, C_j, stride*k+m)$$

如果 `padding` 不是0，会在输入的每一边添加相应数目0 `dilation` 用于控制内核点之间的距离，详细描述在[这里](#)

参数：

- `kernel_size( int or tuple )` - max pooling的窗口大小
- `stride( int or tuple , optional )` - max pooling的窗口移动的步长。默认值是 `kernel_size`
- `padding( int or tuple , optional )` - 输入的每一条边补充0的层数
- `dilation( int or tuple , optional )` - 一个控制窗口中元素步幅的参数
- `return_indices` - 如果等于 `True` ，会返回输出最大值的序号，对于上采样操作会有帮助
- `ceil_mode` - 如果等于 `True` ，计算输出信号大小的时候，会使用向上取整，代替默认的向下取整的操作

**shape:** 输入: (N,C<sub>in</sub>,L<sub>in</sub>) 输出: (N,C<sub>out</sub>,L<sub>out</sub>)

$$L_{out} = \text{floor}((L_{in} + 2padding - dilation(kernel\_size - 1) - 1)/stride + 1)$$

**example:**

```
>>> # pool of size=3, stride=2
>>> m = nn.MaxPool1d(3, stride=2)
>>> input = autograd.Variable(torch.randn(20, 16, 50))
>>> output = m(input)
```

**class torch.nn.MaxPool2d(kernel\_size, stride=None, padding=0, dilation=1, return\_indices=False, ceil\_mode=False)**

对于输入信号的输入通道，提供2维最大池化（max pooling）操作

如果输入的大小是(N,C,H,W)，那么输出的大小是(N,C,H\_out,W\_out)和池化窗口大小(kH,kW)的关系是：

$$out(N_i, C_j, k) = \max_{m=0}^{kH-1} \max_{n=0}^{kW-1} input(N_i, C_j, stride[0]_h+m, stride[1]_w+n)$$

如果 padding 不是0，会在输入的每一边添加相应数目0 dilation 用于控制内核点之间的距离，详细描述在[这里](#)

参数 kernel\_size, stride, padding, dilation 数据类型：可以是一个 int 类型的数据，此时卷积height和width值相同；也可以是一个 tuple 数组（包含来两个int类型的数据），第一个 int 数据表示height的数值，tuple 的第二个int类型的数据表示width的数值

参数：

- kernel\_size( int or tuple ) - max pooling的窗口大小
- stride( int or tuple, optional ) - max pooling的窗口移动的步长。默认值是 kernel\_size
- padding( int or tuple, optional ) - 输入的每一条边补充0的层数
- dilation( int or tuple, optional ) - 一个控制窗口中元素步幅的参数
- return\_indices - 如果等于 True，会返回输出最大值的序号，对于上采样操作会有帮助
- ceil\_mode - 如果等于 True，计算输出信号大小的时候，会使用向上取整，代替默认的向下取整的操作

**shape:** 输入: (N,C,H\_in,W\_in) 输出: (N,C,H\_out,W\_out)

$$H_{out} = \text{floor}((H_{in} + 2padding[0] - dilation[0](kernel\_size[0] - 1) - 1)/stride[0] + 1)$$

$$W_{out} = \text{floor}((W_{in} + 2padding[1] - dilation[1](kernel\_size[1] - 1) - 1)/stride[1] + 1)$$

**example:**

```
>>> # pool of square window of size=3, stride=2
>>> m = nn.MaxPool2d(3, stride=2)
>>> # pool of non-square window
>>> m = nn.MaxPool2d((3, 2), stride=(2, 1))
>>> input = autograd.Variable(torch.randn(20, 16, 50, 32))
>>> output = m(input)
```

## **class torch.nn.MaxPool3d(kernel\_size, stride=None, padding=0, dilation=1, return\_indices=False, ceil\_mode=False)**

对于输入信号的输入通道，提供3维最大池化（max pooling）操作

如果输入的大小是(N,C,D,H,W)，那么输出的大小是(N,C,D,H\_out,W\_out)和池化窗口大小(kD,kH,kW)的关系是：

$$out(N_i, C_j, d, h, w) = \max^{kD-1}_{m=0} \max^{kH-1}_{m=0} \max^{kW-1}_{m=0} input(N_{\{i\}}, C_j, stride[0]_k+d, stride[1]_h+m, stride[2]*w+n)$$

$$input(N_{\{i\}}, C_j, stride[0]_k+d, stride[1]_h+m, stride[2]*w+n)$$

如果 padding 不是0，会在输入的每一边添加相应数目0 dilation 用于控制内核点之间的距离，详细描述在[这里](#)

参数 kernel\_size, stride, padding, dilation 数据类型：可以是 int 类型的数据，此时卷积height和width值相同；也可以是一个 tuple 数组（包含来两个 int 类型的数据），第一个 int 数据表示height的数值，tuple 的第二个 int 类型的数据表示width的数值

参数：

- kernel\_size( int or tuple ) - max pooling的窗口大小
- stride( int or tuple , optional ) - max pooling的窗口移动的步长。默认值是kernel\_size
- padding( int or tuple , optional ) - 输入的每一条边补充0的层数
- dilation( int or tuple , optional ) - 一个控制窗口中元素步幅的参数
- return\_indices - 如果等于 True ，会返回输出最大值的序号，对于上采样操作会有帮助
- ceil\_mode - 如果等于 True ，计算输出信号大小的时候，会使用向上取整，代替默认的向下取整的操作

**shape:** 输入: (N,C,H\_in,W\_in) 输出: (N,C,H\_out,W\_out)

$$D_{out} = \text{floor}((D_{in} + 2padding[0] - dilation[0](kernel\_size[0] - 1) - 1)/stride[0] + 1)$$

$$H_{out} = \text{floor}((H_{in} + 2padding[1] - dilation[1](kernel\_size[0] - 1) - 1)/stride[1] + 1)$$

$$W_{out} = \text{floor}((W_{in} + 2padding[2] - dilation[2](kernel\_size[2] - 1) - 1) / stride[2] + 1)$$

**example:**

```
>>> # pool of square window of size=3, stride=2
>>> m = nn.MaxPool3d(3, stride=2)
>>> # pool of non-square window
>>> m = nn.MaxPool3d((3, 2, 2), stride=(2, 1, 2))
>>> input = autograd.Variable(torch.randn(20, 16, 50, 44, 31))
>>> output = m(input)
```

## class torch.nn.MaxUnpool1d(kernel\_size, stride=None, padding=0)

Maxpool1d 的逆过程，不过并不是完全的逆过程，因为在 maxpool1d 的过程中，一些最大值的已经丢失。MaxUnpool1d 输入 MaxPool1d 的输出，包括最大值的索引，并计算所有 maxpool1d 过程中非最大值被设置为零的部分的反向。

注意：MaxPool1d 可以将多个输入大小映射到相同的输出大小。因此，反演过程可能会变得模棱两可。为了适应这一点，可以在调用中将输出大小（output\_size）作为额外的参数传入。具体用法，请参阅下面的输入和示例

参数：

- kernel\_size( int or tuple ) - max pooling的窗口大小
- stride( int or tuple , optional ) - max pooling的窗口移动的步长。默认值是 kernel\_size
- padding( int or tuple , optional ) - 输入的每一条边补充0的层数

输入：input :需要转换的 tensor indices : Maxpool1d的索引号  
output\_size :一个指定输出大小的 torch.Size

**shape:** input : (N,C,H\_in) output : (N,C,H\_out)

$H_{out} = (H_{in} - 1) \cdot stride[0] - 2 \cdot padding[0] + kernel\_size[0]$  也可以使用 output\_size 指定输出的大小

**Example :**

```
>>> pool = nn.MaxPool1d(2, stride=2, return_indices=True)
>>> unpool = nn.MaxUnpool1d(2, stride=2)
>>> input = Variable(torch.Tensor([[[1, 2, 3, 4, 5, 6, 7, 8]]]))
>>> output, indices = pool(input)
>>> unpool(output, indices)
Variable containing:
(0 ,...,) =
 0 2 0 4 0 6 0 8
[torch.FloatTensor of size 1x1x8]

>>> # Example showcasing the use of output_size
>>> input = Variable(torch.Tensor([[[1, 2, 3, 4, 5, 6, 7, 8, 9]]
]))
>>> output, indices = pool(input)
>>> unpool(output, indices, output_size=input.size())
Variable containing:
(0 ,...,) =
 0 2 0 4 0 6 0 8 0
[torch.FloatTensor of size 1x1x9]
>>> unpool(output, indices)
Variable containing:
(0 ,...,) =
 0 2 0 4 0 6 0 8
[torch.FloatTensor of size 1x1x8]
```

## class torch.nn.MaxUnpool2d(kernel\_size, stride=None, padding=0)

Maxpool2d 的逆过程，不过并不是完全的逆过程，因为在maxpool2d的过程中，一些最大值的已经丢失。 MaxUnpool2d 的输入是 MaxPool2d 的输出，包括最大值的索引，并计算所有 maxpool2d 过程中非最大值被设置为零的部分的反向。

注意： MaxPool2d 可以将多个输入大小映射到相同的输出大小。因此，反演过程可能会变得模棱两可。为了适应这一点，可以在调用中将输出大小（ output\_size ）作为额外的参数传入。具体用法，请参阅下面示例

参数：

- kernel\_size( int or tuple ) - max pooling的窗口大小
- stride( int or tuple , optional ) - max pooling的窗口移动的步长。默认值是 kernel\_size
- padding( int or tuple , optional ) - 输入的每一条边补充0的层数

输入： input :需要转换的 tensor indices : Maxpool1d的索引号  
output\_size :一个指定输出大小的 torch.Size

大小： input : (N,C,H\_in,W\_in) output : (N,C,H\_out,W\_out)

$$H_{out} = (H_{in} - 1) \cdot stride[0] - 2 \cdot padding[0] + kernel\_size[0]$$

$W\{out\} = (W\{in\} - 1) \cdot stride[1] - 2 \cdot padding[1] + kernel\_size[1]$

也可以使用 `output_size` 指定输出的大小

**Example :**

```
>>> pool = nn.MaxPool2d(2, stride=2, return_indices=True)
>>> unpool = nn.MaxUnpool2d(2, stride=2)
>>> input = Variable(torch.Tensor([[[[1, 2, 3, 4],
... [5, 6, 7, 8],
... [9, 10, 11, 12],
... [13, 14, 15, 16]]]])))
>>> output, indices = pool(input)
>>> unpool(output, indices)
Variable containing:
(0 ,0 ,...) =
 0 0 0 0
 0 6 0 8
 0 0 0 0
 0 14 0 16
[torch.FloatTensor of size 1x1x4x4]

>>> # specify a different output size than input size
>>> unpool(output, indices, output_size=torch.Size([1, 1, 5, 5])
)
Variable containing:
(0 ,0 ,...) =
 0 0 0 0 0
 6 0 8 0 0
 0 0 0 14 0
 16 0 0 0 0
 0 0 0 0 0
[torch.FloatTensor of size 1x1x5x5]
```

## class torch.nn.MaxUnpool3d(kernel\_size, stride=None, padding=0)

`Maxpool3d` 的逆过程，不过并不是完全的逆过程，因为在 `maxpool3d` 的过程中，一些最大值的已经丢失。`MaxUnpool3d` 的输入就是 `MaxPool3d` 的输出，包括最大值的索引，并计算所有 `maxpool3d` 过程中非最大值被设置为零的部分的反向。

注意：`MaxPool3d` 可以将多个输入大小映射到相同的输出大小。因此，反演过程可能会变得模棱两可。为了适应这一点，可以在调用中将输出大小（`output_size`）作为额外的参数传入。具体用法，请参阅下面的输入和示例

参数：

- `kernel_size` (int or tuple) - Maxpooling窗口大小
- `stride` (int or tuple, optional) - max pooling的窗口移动的步长。默认



值是 `kernel_size`

- `padding( int or tuple , optional )` - 输入的每一条边补充0的层数

输入： `input` :需要转换的 `tensor`   `indices` : `Maxpool1d` 的索引序数  
`output_size` :一个指定输出大小的 `torch.Size`

大小： `input` :  $(N, C, D_{in}, H_{in}, W_{in})$    `output` :  $(N, C, D_{out}, H_{out}, W_{out})$

$$\begin{aligned} D_{out} &= (D_{in} - 1) \cdot stride[0] - 2 \cdot padding[0] + kernel\_size[0] \\ H_{out} &= (H_{in} - 1) \cdot stride[1] - 2 \cdot padding[0] + kernel\_size[1] \\ W_{out} &= (W_{in} - 1) \cdot stride[2] - 2 \cdot padding[2] + kernel\_size[2] \end{aligned}$$

也可以使用 `output_size` 指定输出的大小

**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])
```

## **`class torch.nn.AvgPool1d(kernel_size, stride=None, padding=0, ceil_mode=False, count_include_pad=True)`**

对信号的输入通道，提供1维平均池化（average pooling）输入信号的大小  $(N, C, L)$ ，输出大小  $(N, C, L_{out})$  和池化窗口大小  $k$  的关系是：

$out(N, C, l) = 1/k * \sum_{m=0}^{k-1} input(N, C, l + m)$  如果 `padding` 不是 0，会在输入的每一边添加相应数目 0

参数：

- `kernel_size( int or tuple )` - 池化窗口大小
- `stride( int or tuple , optional )` - max pooling 的窗口移动的步长。默认值是 `kernel_size`
- `padding( int or tuple , optional )` - 输入的每一条边补充0的层数
- `dilation( int or tuple , optional )` - 一个控制窗口中元素步幅的参数
- `return_indices` - 如果等于 `True`，会返回输出最大值的序号，对于上采样操作会有帮助
- `ceil_mode` - 如果等于 `True`，计算输出信号大小的时候，会使用向上取整，代替默认的向下取整的操作

大小： `input` :  $(N, C, L_{in})$    `output` :  $(N, C, L_{out})$

$$L_{out} = \text{floor}((L_{in} + 2 * padding - kernel\_size) / stride + 1)$$

**Example:**

```
>>> # pool with window of size=3, stride=2
>>> m = nn.AvgPool1d(3, stride=2)
>>> m(Variable(torch.Tensor([[[1, 2, 3, 4, 5, 6, 7]]]])))
Variable containing:
 (0 , ..) =
 2 4 6
 [torch.FloatTensor of size 1x1x3]
```

## **class torch.nn.AvgPool2d(kernel\_size, stride=None, padding=0, ceil\_mode=False, count\_include\_pad=True)**

对信号的输入通道，提供2维的平均池化（average pooling）输入信号的大小(N,C,H,W)，输出大小(N,C,H\_out,W\_out)和池化窗口大小(kH,kW)的关系是：

$$out(N_i, C_j, h, w) = \frac{1}{(kH \cdot kW)} \sum_{m=0}^{kH-1} \sum_{n=0}^{kW-1} input(N_i, C_j, stride[0]_h + m, stride[1]_w + n)$$

如果 padding 不是0，会在输入的每一边添加相应数目0

参数：

- kernel\_size( int or tuple ) - 池化窗口大小
- stride( int or tuple , optional ) - max pooling的窗口移动的步长。默认值是 kernel\_size
- padding( int or tuple , optional ) - 输入的每一条边补充0的层数
- dilation( int or tuple , optional ) - 一个控制窗口中元素步幅的参数
- ceil\_mode - 如果等于 True ，计算输出信号大小的时候，会使用向上取整，代替默认的向下取整的操作
- count\_include\_pad - 如果等于 True ，计算平均池化时，将包括 padding 填充的0

**shape :** input : (N,C,H\_in,W\_in) output : (N,C,H\_out,W\_out)

$$\begin{aligned} H_{out} &= \text{floor}((H_{in} + 2padding[0] - kernel\_size[0]) / stride[0] + 1) \\ W_{out} &= \text{floor}((W_{in} + 2padding[1] - kernel\_size[1]) / stride[1] + 1) \end{aligned}$$

**Example:**

```
>>> # pool of square window of size=3, stride=2
>>> m = nn.AvgPool2d(3, stride=2)
>>> # pool of non-square window
>>> m = nn.AvgPool2d((3, 2), stride=(2, 1))
>>> input = autograd.Variable(torch.randn(20, 16, 50, 32))
>>> output = m(input)
```



## class torch.nn.AvgPool3d(kernel\_size, stride=None)

对信号的输入通道，提供3维的平均池化（average pooling）输入信号的大小(N,C,D,H,W)，输出大小(N,C,D\_out,H\_out,W\_out)和池化窗口大小(kD,kH,kW)的关系是：

$$\text{out}(N_i, C_j, d, h, w) = \frac{1}{(kD \cdot kH \cdot kW)} \sum_{k=0}^{kD-1} \sum_{h=0}^{kH-1} \sum_{w=0}^{kW-1} \text{input}(N\{i\}, C\{j\}, \text{stride}[0]_d + k, \text{stride}[1]_h + m, \text{stride}[2]_w + n)$$

如果 padding 不是0，会在输入的每一边添加相应数目0

参数：

- kernel\_size( int or tuple ) - 池化窗口大小
- stride( int or tuple , optional ) - max pooling 的窗口移动的步长。  
默认值是 kernel\_size

**shape**：输入大小:(N,C,D<sub>in</sub>,H<sub>in</sub>,W<sub>in</sub>) 输出大小:(N,C,D<sub>out</sub>,H<sub>out</sub>,W<sub>out</sub>)

$$D\{out\} = \text{floor}((D\{in\} + 2 * \text{padding}[0] - \text{kernel\_size}[0]) / \text{stride}[0] + 1)$$

$$H\{out\} = \text{floor}((H\{in\} + 2 * \text{padding}[1] - \text{kernel\_size}[1]) / \text{stride}[1] + 1)$$

$$W\{out\} = \text{floor}((W\{in\} + 2 * \text{padding}[2] - \text{kernel\_size}[2]) / \text{stride}[2] + 1)$$

**Example:**

```
>>> # pool of square window of size=3, stride=2
>>> m = nn.AvgPool3d(3, stride=2)
>>> # pool of non-square window
>>> m = nn.AvgPool3d((3, 2, 2), stride=(2, 1, 2))
>>> input = autograd.Variable(torch.randn(20, 16, 50, 44, 31))
>>> output = m(input)
```

## class torch.nn.FractionalMaxPool2d(kernel\_size, output\_size=None, output\_ratio=None, return\_indices=False, \_random\_samples=None)

对输入的信号，提供2维的分数最大化池化操作 分数最大化池化的细节请阅读[论文](#) 由目标输出大小确定的随机步长,在 $kH \times kW$ 区域进行最大池化操作。输出特征和输入特征的数量相同。

参数：

- kernel\_size( int or tuple ) - 最大池化操作时的窗口大小。可以是一个数字（表示  $K \times K$  的窗口），也可以是一个元组（  $kh \times kw$  ）
- output\_size - 输出图像的尺寸。可以使用一个 tuple 指定(oH,oW)，也可以使用一个数字oH指定一个oH\*oH的输出。
- output\_ratio - 将输入图像的大小的百分比指定为输出图片的大小，使用一个范围在(0,1)之间的数字指定
- return\_indices - 默认值 False，如果设置为 True，会返回输出的索引，索引对 nn.MaxUnpool2d 有用。

**Example :**

```
>>> # pool of square window of size=3, and target output size 13
x12
>>> m = nn.FractionalMaxPool2d(3, output_size=(13, 12))
>>> # pool of square window and target output size being half of
input image size
>>> m = nn.FractionalMaxPool2d(3, output_ratio=(0.5, 0.5))
>>> input = autograd.Variable(torch.randn(20, 16, 50, 32))
>>> output = m(input)
```

## class torch.nn.LPPool2d(norm\_type, kernel\_size, stride=None, ceil\_mode=False)

对输入信号提供2维的幂平均池化操作。输出的计算方式：

$$f(x) = \text{pow}(\text{sum}(X, p), 1/p)$$

- 当  $p$  为无穷大的时候时，等价于最大池化操作
- 当  $p=1$  时，等价于平均池化操作

参数 `kernel_size` , `stride` 的数据类型：

- `int` , 池化窗口的宽和高相等
- `tuple` 数组（两个数字的），一个元素是池化窗口的高，另一个是宽

参数

- `kernel_size`: 池化窗口的大小
- `stride`: 池化窗口移动的步长。 `kernel_size` 是默认值
- `ceil_mode`: `ceil_mode=True` 时，将使用向下取整代替向上取整

**shape**

- 输入：(N,C,H<sub>in</sub>,W<sub>in</sub>)
- 输出：(N,C,H<sub>out</sub>,W<sub>out</sub>) 
$$\begin{aligned} H_{out} &= \text{floor}((H_{in} + 2\_padding[0] - dilation[0](kernel\_size[0] - 1) - 1) / stride[0] + 1) \\ W_{out} &= \text{floor}((W_{in} + 2\_padding[1] - dilation[1](kernel\_size[1] - 1) - 1) / stride[1] + 1) \end{aligned}$$

**Example:**

```
>>> # power-2 pool of square window of size=3, stride=2
>>> m = nn.LPPool2d(2, 3, stride=2)
>>> # pool of non-square window of power 1.2
>>> m = nn.LPPool2d(1.2, (3, 2), stride=(2, 1))
>>> input = autograd.Variable(torch.randn(20, 16, 50, 32))
>>> output = m(input)
```

## class torch.nn.AdaptiveMaxPool1d(output\_size, return\_indices=False)

对输入信号，提供1维的自适应最大池化操作 对于任何输入大小的输入，可以将输出尺寸指定为H，但是输入和输出特征的数目不会变化。

参数：

- output\_size: 输出信号的尺寸
- return\_indices: 如果设置为 True ，会返回输出的索引。对 nn.MaxUnpool1d 有用，默认值是 False

Example：

```
>>> # target output size of 5
>>> m = nn.AdaptiveMaxPool1d(5)
>>> input = autograd.Variable(torch.randn(1, 64, 8))
>>> output = m(input)
```

## class torch.nn.AdaptiveMaxPool2d(output\_size, return\_indices=False)

对输入信号，提供2维的自适应最大池化操作 对于任何输入大小的输入，可以将输出尺寸指定为H\*W，但是输入和输出特征的数目不会变化。

参数：

- output\_size: 输出信号的尺寸,可以用 (H,W) 表示 H\*W 的输出，也可以使用数字 H 表示 H\*H 大小的输出
- return\_indices: 如果设置为 True ，会返回输出的索引。对 nn.MaxUnpool2d 有用，默认值是 False

Example：

```
>>> # target output size of 5x7
>>> m = nn.AdaptiveMaxPool2d((5,7))
>>> input = autograd.Variable(torch.randn(1, 64, 8, 9))
>>> # target output size of 7x7 (square)
>>> m = nn.AdaptiveMaxPool2d(7)
>>> input = autograd.Variable(torch.randn(1, 64, 10, 9))
>>> output = m(input)
```

## class torch.nn.AdaptiveAvgPool1d(output\_size)

对输入信号，提供1维的自适应平均池化操作 对于任何输入大小的输入，可以将输出尺寸指定为H\*W，但是输入和输出特征的数目不会变化。

参数：

- `output_size`: 输出信号的尺寸

**Example :**

```
>>> # target output size of 5
>>> m = nn.AdaptiveAvgPool1d(5)
>>> input = autograd.Variable(torch.randn(1, 64, 8))
>>> output = m(input)
```

## class torch.nn.AdaptiveAvgPool2d(output\_size)

对输入信号，提供2维的自适应平均池化操作 对于任何输入大小的输入，可以将输出尺寸指定为 `H*W`，但是输入和输出特征的数目不会变化。

参数：

- `output_size`: 输出信号的尺寸,可以用(H,W)表示 `H*W` 的输出，也可以使用耽搁数字H表示H\*H大小的输出

**Example :**

```
>>> # target output size of 5x7
>>> m = nn.AdaptiveAvgPool2d((5,7))
>>> input = autograd.Variable(torch.randn(1, 64, 8, 9))
>>> # target output size of 7x7 (square)
>>> m = nn.AdaptiveAvgPool2d(7)
>>> input = autograd.Variable(torch.randn(1, 64, 10, 9))
>>> output = m(input)
```

## Non-Linear Activations

```
class torch.nn.ReLU(inplace=False)
```

对输入运用修正线性单元函数 $\text{ReLU}(x) = \max(0, x)$ ，

参数：`inplace`-选择是否进行覆盖运算

shape：

- 输入： $(N, )$ ，代表任意数目附加维度
- 输出： $(N, *)$ ，与输入拥有同样的shape属性

例子：

```
>>> m = nn.ReLU()
>>> input = autograd.Variable(torch.randn(2))
>>> print(input)
>>> print(m(input))
```

```
class torch.nn.ReLU6(inplace=False)
```

对输入的每一个元素运用函数 $\text{ReLU6}(x) = \min(\max(0, x), 6)$ ，

参数：inplace-选择是否进行覆盖运算

shape：

- 输入： $(N, *)$ ，代表任意数目附加维度
- 输出： $(N, *)$ ，与输入拥有同样的shape属性

例子：

```
>>> m = nn.ReLU6()
>>> input = autograd.Variable(torch.randn(2))
>>> print(input)
>>> print(m(input))
```

```
class torch.nn.ELU(alpha=1.0, inplace=False)
```

对输入的每一个元素运用函数 $f(x) = \max(0, x) + \min(0, \alpha * (e^x - 1))$ ，

shape：

- 输入： $(N, *)$ ，星号代表任意数目附加维度
- 输出： $(N, *)$ 与输入拥有同样的shape属性

例子：

```
>>> m = nn.ELU()
>>> input = autograd.Variable(torch.randn(2))
>>> print(input)
>>> print(m(input))
```

```
class torch.nn.PReLU(num_parameters=1, init=0.25)
```

对输入的每一个元素运用函数 $\text{PReLU}(x) = \max(0, x) + a * \min(0, x)$ ， $a$  是一个可学习参数。当没有声明时，`nn.PReLU()` 在所有的输入中只有一个参数  $a$ ；如果是 `nn.PReLU(nChannels)`， $a$  将应用到每个输入。

注意：当为了表现更佳的模型而学习参数  $a$  时不要使用权重衰减（weight decay）

参数：

- `num_parameters` : 需要学习的 `a` 的个数，默认等于1
- `init` : `a` 的初始值，默认等于0.25

shape :

- 输入 :  $(N, )$  , 代表任意数目附加维度
- 输出 :  $(N, *)$  , 与输入拥有同样的shape属性

例子 :

```
>>> m = nn.PReLU()
>>> input = autograd.Variable(torch.randn(2))
>>> print(input)
>>> print(m(input))
```

```
class torch.nn.LeakyReLU(negative_slope=0.01, inplace=False)
```

对输入的每一个元素运用  $f(x) = \max(0, x) + \{\text{negative\_slope}\} * \min(0, x)$

参数 :

- `negative_slope` : 控制负斜率的角度，默认等于0.01
- `inplace` - 选择是否进行覆盖运算

shape :

- 输入 :  $(N, )$  , 代表任意数目附加维度
- 输出 :  $(N, *)$  , 与输入拥有同样的shape属性

例子 :

```
>>> m = nn.LeakyReLU(0.1)
>>> input = autograd.Variable(torch.randn(2))
>>> print(input)
>>> print(m(input))
```

```
class torch.nn.Threshold(threshold, value, inplace=False)
```

Threshold定义 :

$y = x$  , if  $x \geq \text{threshold}$   $y = \text{value}$  , if  $x < \text{threshold}$

参数 :

- `threshold` : 阈值
- `value` : 输入值小于阈值则会被value代替
- `inplace` : 选择是否进行覆盖运算

shape :

- 输入 :  $(N, )$  , 代表任意数目附加维度

- 输出： $(N, *)$ ，与输入拥有同样的shape属性

例子：

```
>>> m = nn.Threshold(0.1, 20)
>>> input = Variable(torch.randn(2))
>>> print(input)
>>> print(m(input))
```

```
class torch.nn.Hardtanh(min_value=-1, max_value=1, inplace=False)
```

对每个元素，

$f(x) = +1, \text{ if } x > 1; f(x) = -1, \text{ if } x < -1; f(x) = x, \text{ otherwise}$

线性区域的范围 $[-1, 1]$ 可以被调整

参数：

- min\_value：线性区域范围最小值
- max\_value：线性区域范围最大值
- inplace：选择是否进行覆盖运算

shape：

- 输入： $(N, )$ ，表示任意维度组合
- 输出： $(N, *)$ ，与输入有相同的shape属性

例子：

```
>>> m = nn.Hardtanh()
>>> input = autograd.Variable(torch.randn(2))
>>> print(input)
>>> print(m(input))
```

```
class torch.nn.Sigmoid
```

对每个元素运用Sigmoid函数，Sigmoid 定义如下：

$f(x) = 1 / (1 + e^{-x})$

shape：

- 输入： $(N, )$ ，表示任意维度组合
- 输出： $(N, *)$ ，与输入有相同的shape属性

例子：

```
>>> m = nn.Sigmoid()
>>> input = autograd.Variable(torch.randn(2))
>>> print(input)
>>> print(m(input))
```

`class torch.nn.Tanh`

对输入的每个元素，

$$f(x) = \frac{e^x - e^{-x}}{e^x + e^{-x}}$$

shape :

- 输入：(N, )，表示任意维度组合
- 输出：(N, \*)，与输入有相同的shape属性

例子：

```
>>> m = nn.Tanh()
>>> input = autograd.Variable(torch.randn(2))
>>> print(input)
>>> print(m(input))
```

`class torch.nn.LogSigmoid`

对输入的每个元素， $\text{LogSigmoid}(x) = \log(1 / (1 + e^{-x}))$

shape :

- 输入：(N, )，表示任意维度组合
- 输出：(N, \*)，与输入有相同的shape属性

例子：

```
>>> m = nn.LogSigmoid()
>>> input = autograd.Variable(torch.randn(2))
>>> print(input)
>>> print(m(input))
```

`class torch.nn.Softplus(beta=1, threshold=20)`

对每个元素运用Softplus函数，Softplus 定义如下：

$$f(x) = \frac{1}{\beta} \log(1 + e^{\beta x_i})$$

Softplus函数是ReLU函数的平滑逼近，Softplus函数可以使得输出值限定为正数。

为了保证数值稳定性，线性函数的转换可以使输出大于某个值。

参数：



- `beta` : Softplus函数的`beta`值
- `threshold` : 阈值

shape :

- 输入 : (N, ), 表示任意维度组合
- 输出 : (N, \*), 与输入有相同的shape属性

例子 :

```
>>> m = nn.Softplus()
>>> input = autograd.Variable(torch.randn(2))
>>> print(input)
>>> print(m(input))
```

`class torch.nn.Softshrink(lambd=0.5)`

对每个元素运用Softshrink函数，Softshrink函数定义如下：

$f(x) = x - \lambda$ , if  $x > \lambda$   $f(x) = x + \lambda$ , if  $x < -\lambda$   $f(x) = 0$ , otherwise

参数：

`lambd` : Softshrink函数的`lambda`值，默认为0.5

shape :

- 输入 : (N, ), 表示任意维度组合
- 输出 : (N, \*), 与输入有相同的shape属性

例子 :

```
>>> m = nn.Softshrink()
>>> input = autograd.Variable(torch.randn(2))
>>> print(input)
>>> print(m(input))
```

`class torch.nn.Softsign`

$f(x) = x / (1 + |x|)$

shape :

- 输入 : (N, ), 表示任意维度组合
- 输出 : (N, \*), 与输入有相同的shape属性

例子 :

```
>>> m = nn.Softsign()
>>> input = autograd.Variable(torch.randn(2))
>>> print(input)
>>> print(m(input))
```

`class torch.nn.Softshrink(lambd=0.5)`

对每个元素运用Tanhshrink函数，Tanhshrink函数定义如下：

$\text{Tanhshrink}(x) = x - \text{Tanh}(x)$

shape：

- 输入：(N, )，表示任意维度组合
- 输出：(N, \*)，与输入有相同的shape属性

例子：

```
>>> m = nn.Tanhshrink()
>>> input = autograd.Variable(torch.randn(2))
>>> print(input)
>>> print(m(input))
```

`class torch.nn.Softmin`

对n维输入张量运用Softmin函数，将张量的每个元素缩放到（0,1）区间且和为1。Softmin函数定义如下：

$$\text{Softmin}_i(x) = \frac{e^{-(x_i - \text{shift})}}{\sum_j e^{-(x_j - \text{shift})}}, \text{shift} = \max(x_i)$$

shape：

- 输入：(N, L)
- 输出：(N, L)

例子：

```
>>> m = nn.Softmin()
>>> input = autograd.Variable(torch.randn(2, 3))
>>> print(input)
>>> print(m(input))
```

`class torch.nn.Softmax`

对n维输入张量运用Softmax函数，将张量的每个元素缩放到（0,1）区间且和为1。Softmax函数定义如下：

$$f_i(x) = \frac{e^{(x_i - \text{shift})}}{\sum_j e^{(x_j - \text{shift})}}, \text{shift} = \max(x_i)$$

shape :

- 输入：(N, L)
- 输出：(N, L)

返回结果是一个与输入维度相同的张量，每个元素的取值范围在（0,1）区间。

例子：

```
>>> m = nn.Softmax()
>>> input = autograd.Variable(torch.randn(2, 3))
>>> print(input)
>>> print(m(input))
```

`class torch.nn.LogSoftmax`

对n维输入张量运用LogSoftmax函数，LogSoftmax函数定义如下：

$$f_i(x) = \log \frac{e^{(x_i)}}{a}, a = \sum_j e^{(x_j)}$$

shape :

- 输入：(N, L)
- 输出：(N, L)

例子：

```
>>> m = nn.LogSoftmax()
>>> input = autograd.Variable(torch.randn(2, 3))
>>> print(input)
>>> print(m(input))
```

## Normalization layers

**`class torch.nn.BatchNorm1d(num_features, eps=1e-05, momentum=0.1, affine=True)`**

对小批量(mini-batch)的2d或3d输入进行批标准化(Batch Normalization)操作

$$y = \frac{x - \text{mean}[x]}{\sqrt{\text{Var}[x] + \epsilon}} * \gamma + \beta$$

在每一个小批量（mini-batch）数据中，计算输入各个维度的均值和标准差。  
gamma与beta是可学习的大小为C的参数向量（C为输入大小）

在训练时，该层计算每次输入的均值与方差，并进行移动平均。移动平均默认的动量值为0.1。

在验证时，训练求得的均值/方差将用于标准化验证数据。

参数：

- **num\_features**：来自期望输入的特征数，该期望输入的大小为'batch\_size x num\_features [x width]'
- **eps**：为保证数值稳定性（分母不能趋近或取0），给分母加上的值。默认为1e-5。
- **momentum**：动态均值和动态方差所使用的动量。默认为0.1。
- **affine**：一个布尔值，当设为true，给该层添加可学习的仿射变换参数。

Shape：

- 输入：(N, C) 或者 (N, C, L)
- 输出：(N, C) 或者 (N, C, L)（输入输出相同）

例子

```
>>> # With Learnable Parameters
>>> m = nn.BatchNorm1d(100)
>>> # Without Learnable Parameters
>>> m = nn.BatchNorm1d(100, affine=False)
>>> input = autograd.Variable(torch.randn(20, 100))
>>> output = m(input)
```

## class torch.nn.BatchNorm2d(num\_features, eps=1e-05, momentum=0.1, affine=True)

对小批量(mini-batch)3d数据组成的4d输入进行批标准化(Batch Normalization)操作

$$y = \frac{x - \text{mean}[x]}{\sqrt{\text{Var}[x] + \epsilon}} * \gamma + \beta$$

在每一个小批量（mini-batch）数据中，计算输入各个维度的均值和标准差。  
gamma与beta是可学习的大小为C的参数向量（C为输入大小）

在训练时，该层计算每次输入的均值与方差，并进行移动平均。移动平均默认的动量值为0.1。

在验证时，训练求得的均值/方差将用于标准化验证数据。

参数：

- **num\_features**：来自期望输入的特征数，该期望输入的大小为'batch\_size x num\_features x height x width'
- **eps**：为保证数值稳定性（分母不能趋近或取0），给分母加上的值。默认为1e-5。
- **momentum**：动态均值和动态方差所使用的动量。默认为0.1。
- **affine**：一个布尔值，当设为true，给该层添加可学习的仿射变换参数。

**Shape :**

- 输入：(N, C, H, W)
- 输出：(N, C, H, W) (输入输出相同)

例子

```
>>> # With Learnable Parameters
>>> m = nn.BatchNorm2d(100)
>>> # Without Learnable Parameters
>>> m = nn.BatchNorm2d(100, affine=False)
>>> input = autograd.Variable(torch.randn(20, 100, 35, 45))
>>> output = m(input)
```

## **class torch.nn.BatchNorm3d(num\_features, eps=1e-05, momentum=0.1, affine=True)**

对小批量(mini-batch)4d数据组成的5d输入进行批标准化(Batch Normalization)操作

$$y = \frac{x - \text{mean}[x]}{\sqrt{\text{Var}[x] + \epsilon}} * \gamma + \beta$$

在每一个小批量 (mini-batch) 数据中，计算输入各个维度的均值和标准差。  
gamma与beta是可学习的大小为C的参数向量 (C为输入大小)

在训练时，该层计算每次输入的均值与方差，并进行移动平均。移动平均默认的动量值为0.1。

在验证时，训练求得的均值/方差将用于标准化验证数据。

参数：

- **num\_features**：来自期望输入的特征数，该期望输入的大小为'batch\_size x num\_features depth x height x width'
- **eps**：为保证数值稳定性（分母不能趋近或取0），给分母加上的值。默认为1e-5。
- **momentum**：动态均值和动态方差所使用的动量。默认为0.1。
- **affine**：一个布尔值，当设为true，给该层添加可学习的仿射变换参数。

**Shape :**

- 输入：(N, C, D, H, W)
- 输出：(N, C, D, H, W) (输入输出相同)

例子

```
>>> # With Learnable Parameters
>>> m = nn.BatchNorm3d(100)
>>> # Without Learnable Parameters
>>> m = nn.BatchNorm3d(100, affine=False)
>>> input = autograd.Variable(torch.randn(20, 100, 35, 45, 10))
>>> output = m(input)
```

## Recurrent layers

### class torch.nn.RNN( args, \* kwargs)

将一个多层的 Elman RNN，激活函数为 `tanh` 或者 `ReLU`，用于输入序列。

对输入序列中每个元素，RNN 每层的计算公式为  $h_t = \tanh(w_{ih}x_t + b_{ih} + w_{hh}h_{t-1} + b_{hh})$ 。 $h_t$  是时刻  $t$  的隐状态。 $x_t$  是上一层时刻  $t$  的隐状态，或者是第一层在时刻  $t$  的输入。如果 `nonlinearity='relu'`，那么将使用 `relu` 代替 `tanh` 作为激活函数。

参数说明:

- `input_size` – 输入 `x` 的特征数量。
- `hidden_size` – 隐层的特征数量。
- `num_layers` – RNN 的层数。
- `nonlinearity` – 指定非线性函数使用 `tanh` 还是 `relu`。默认是 `tanh`。
- `bias` – 如果是 `False`，那么 RNN 层就不会使用偏置权重 `$b_{ih}$` 和 `$b_{hh}$`，默认是 `True`。
- `batch_first` – 如果 `True` 的话，那么输入 `Tensor` 的 shape 应该是 `[batch_size, time_step, feature]`，输出也是这样。
- `dropout` – 如果值非零，那么除了最后一层外，其它层的输出都会套上一个 `dropout` 层。
- `bidirectional` – 如果 `True`，将会变成一个双向 RNN，默认为 `False`。

RNN 的输入：(`input`, `h_0`)

- `input (seq_len, batch, input_size)`: 保存输入序列特征的 `tensor`。`input` 可以是被填充的变长的序列。细节请看 `torch.nn.utils.rnn.pack_padded_sequence()`
- `h_0 (num_layers * num_directions, batch, hidden_size)`: 保存着初始隐状态的 `tensor`

**RNN 的输出：(output, h\_n)**

- `output (seq_len, batch, hidden_size * num_directions)`: 保存着 RNN 最后一层的输出特征。如果输入是被填充过的序列，那么输出也是被填充的序列。
- `h_n (num_layers * num_directions, batch, hidden_size)`: 保存着最后一个时刻隐状态。

**RNN 模型参数:**

- `weight_ih_l[k]` – 第 `k` 层的 `input-hidden` 权重，可学习，形状是 `(input_size x hidden_size)`。
- `weight_hh_l[k]` – 第 `k` 层的 `hidden-hidden` 权重，可学习，形状是 `(hidden_size x hidden_size)`
- `bias_ih_l[k]` – 第 `k` 层的 `input-hidden` 偏置，可学习，形状是 `(hidden_size)`
- `bias_hh_l[k]` – 第 `k` 层的 `hidden-hidden` 偏置，可学习，形状是 `(hidden_size)`

示例：

```
rnn = nn.RNN(10, 20, 2)
input = Variable(torch.randn(5, 3, 10))
h0 = Variable(torch.randn(2, 3, 20))
output, hn = rnn(input, h0)
```

**class torch.nn.LSTM( args, \*kwargs)**

将一个多层的 (LSTM) 应用到输入序列。

对输入序列的每个元素，LSTM 的每层都会执行以下计算：

$$\begin{aligned} & \begin{bmatrix} i_t = \text{sigmoid}(W_{ii} x_t + b_{ii} + W_{hi} h_{(t-1)} + b_{hi}) \\ f_t = \text{sigmoid}(W_{if} x_t + b_{if} + W_{hf} h_{(t-1)} + b_{hf}) \\ g_t = \tanh(W_{ig} x_t + b_{ig} + W_{hg} h_{(t-1)} + b_{hg}) \\ o_t = \text{sigmoid}(W_{io} x_t + b_{io} + W_{ho} h_{(t-1)} + b_{ho}) \\ c_t = f_t * c_{(t-1)} + i_t * g_t \\ h_t = o_t * \tanh(c_t) \end{bmatrix} \end{aligned}$$
 是时刻  $t$  的隐状态， $c_t$  是时刻  $t$  的细胞状态， $x_t$  是上一层的在时刻  $t$  的隐状态或者是第一层在时刻  $t$  的输入。 $i_t, f_t, g_t, o_t$  分别代表输入门，遗忘门，细胞

`class torch.nn.GRU (*args, **kwargs)`[\[source\]](#)

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:

$$\begin{aligned} & \begin{bmatrix} r_t = \text{sigmoid}(W_{ir} x_t + b_{ir} + W_{hr} h_{(t-1)} + b_{hr}) \\ z_t = \text{sigmoid}(W_{iz} x_t + b_{iz} + W_{hz} h_{(t-1)} + b_{hz}) \\ n_t = \tanh(W_{in} x_t + b_{in} + r_t * (W_{hn} h_{(t-1)} + b_{hn})) \\ h_t = (1 - z_t) n_t + z_t h_{(t-1)} \end{bmatrix} \end{aligned}$$
 where  $(h_t)$  is the hidden state at time  $t$ ,  $(x_t)$  is

the hidden state of the previous layer at time `t` or `(input_t)` for the first layer, and `(r_t)`, `(z_t)`, `(n_t)` are the reset, input, and new gates, respectively. | 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. ***bias*** – If False, then the layer does not use bias weights `b_ih` and `b_hh`. Default: True ***batch\_first*** – If True, then the input and output tensors are provided as  $(batch, seq, feature)$  ***dropout*** – If non-zero, introduc



## torch.nn.functional

### 卷积函数

```
torch.nn.functional.conv1d(input, weight, bias=None, stride=1, padding=0, dilation=1, groups=1) → Tensor
```

#### source

对由几个平面组成的输入进行卷积操作 对于细节和输出形状，详细可见[Conv1d](#)

#### 参数：

`input`：输入的张量形状(minibatch x in\_channels x iW)  
`weight` - 过滤器的形状 (out\_channels, in\_channels, kW)  
`bias` - 可选偏置的形状(out\_channels)默认值：`None`  
`stride` - 卷积内核的步长，默认为1  
`padding` - 输入上的隐含零填充。可以是单个数字或元组。默认值：0  
`dilation` - 内核元素之间的间距。默认值：1  
`groups` - 将输入分成组，in\_channels应该被组数整除。默认值：1

#### 举例：

```
>>> filters = autograd.Variable(torch.randn(33, 16, 3))
>>> inputs = autograd.Variable(torch.randn(20, 16, 50))
>>> F.conv1d(inputs, filters)
```

```
torch.nn.functional.conv2d(input, weight, bias=None, stride=1, padding=0, dilation=1, groups=1) → Tensor
```

#### source

在由几个输入平面组成的输入图像上应用2D卷积。对于细节和输出形状详细可见[Conv2d](#)

#### 参数：

**input** - 输入的张量 (minibatch x in\_channels x iH x iW)  
**weight** - 过滤器 (out\_channels, in\_channels/groups, kH, kW)  
**bias** - 可选偏置张量(out\_channels)。默认值: **None**  
**stride** - 卷积核的步长, 可以是单个数字或元组 (sh x sw)。默认值: **1**  
**padding** - 输入中默认**0**填充。可以是单个数字或元组。默认值:**0**  
**dilation** - 核元素之间的间距。默认值: **1**  
**groups** - 将输入分成组, in\_channels应该被组数整除。默认值: **1**

举例:

```
>>> # 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)
```

```
torch.nn.functional.conv3d(input, weight, bias=None, stride=1, padding=0, dilation=1, groups=1) → Tensor
```

## source

在由几个输入平面组成的输入图像上应用3D卷积。对于细节和输出形状, 查看[Conv3d](#)

参数:

**input** - 输入张量的形状 (minibatch x in\_channels x iT x iH x iW)  
**weight** - 过滤器的形状 (out\_channels x in\_channels/groups x kT x kH x kW)  
**bias** - 可选的偏差项的大小(out\_channels).默认值是: **None**  
**stride** - 卷积核的步长, 可以是单个数字或元组 (st x sh x sw).默认值: **1**  
**padding** - 在输入中默认的**0**填充。可以是单个数字或元组(padT, padH, padW).默认值:**0**  
**dilation** - 核元素之间的间距(dT, dH, dW)。默认值: **1**  
**groups** - 将输入分成组, in\_channels应该被组数整除。默认值: **1**

举例:

```
>>> filters = torch.randn(33, 16, 3, 3, 3)
>>> inputs = torch.randn(20, 16, 50, 10, 20)
>>> F.conv3d(inputs, filters)
```

```
torch.nn.functional.conv_transpose1d(input, weight, bias=None, stride=1, padding=0, output_padding=0, groups=1, dilation=1) → Tensor
```

## source

在由几个输入平面组成的输入图像上应用1D转置卷积，有时也被称为去卷积。有关详细信息和输出形状，参考[ConvTranspose1d](#)。

## 参数：

**input**：输入的张量形状(minibatch x in\_channels x iW)  
**weight** - 过滤器的形状 (in\_channels x out\_channels/groups x kW)  
**bias** - 可选偏置的形状(out\_channels)默认值：[None](#)  
**stride** - 卷积内核的步长，也可以是一个数字或者元组 (sw)，默认为[1](#)  
**padding** - 输入上的隐含零填充( $0 \leq \text{padding} < \text{stride}$ )。可以是单个数字或者元组(padW)。默认值：[0](#)  
**output\_padding**-在两端的进行0填充 ( $0 \leq \text{padding} < \text{stride}$ )，可以是单一数字或者元组(out\_padw)。默认是[0](#)。  
**dilation** - 内核元素之间的间距。可以是单个数字或者元组(dw)。默认值：[1](#)  
**groups** - 将输入分成组，in\_channels应该被组数整除。默认值：[1](#)

## 举例：

```
>>> inputs = torch.randn(20, 16, 50)
>>> weights = torch.randn(16, 33, 5)
>>> F.conv_transpose1d(inputs, weights)
>
```

```
torch.nn.functional.conv_transpose2d(input, weight, bias=None, stride=1, padding=0, output_padding=0, groups=1, dilation=1) → Tensor
```

## source

在由几个输入平面组成的输入图像上应用2D转置卷积，有时也被称为去卷积。有关详细信息和输出形状，参考[ConvTranspose2d](#)。

## 参数：

**input**: 输入的张量形状(minibatch x in\_channels x iH x iW)  
**weight** - 过滤器的形状 (in\_channels x out\_channels/groups x kH x kW)  
**bias** - 可选偏置的形状(out\_channels)默认值: **None**  
**stride** - 卷积内核的步长, 也可以是一个数字或者元组 (sH, sW), 默认为 **1**  
**padding** - 输入上的隐含零填充 ( $0 \leq \text{padding} < \text{stride}$ )。可以是单个数字或者元组 (padH, padW)。默认值: **0**  
**output\_padding**-在两端的进行**0**填充 ( $0 \leq \text{padding} < \text{stride}$ )，可以是单一数字或者元组 (out\_padH, out\_padW)。默认是 **0**。  
**dilation** - 内核元素之间的间距。可以是单个数字或者元组 (dH, dW)。默认值: **1**  
**groups** - 将输入分成组, in\_channels应该被组数整除。默认值: **1**

举例：

```

>>> # 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)

```

```

torch.nn.functional.conv_transpose3d(input, weight, bias=None, s
stride=1, padding=0, output_padding=0, groups=1, dilation=1) → Te
nsor

```

**source** 在由几个输入平面组成的输入图像上应用3D转置卷积，有时也被称为去卷积。有关详细信息和输出形状，参考[ConvTranspose3d](#)。

参数：

**input**: 输入的张量形状(minibatch x in\_channels x iT x iH x iW)  
**weight** - 过滤器的形状 (in\_channels x out\_channels/groups x kT x kH x kW)  
**bias** - 可选偏置的形状(out\_channels)默认值: **None**  
**stride** - 卷积内核的步长, 也可以是一个数字或者元组 (sT, sH, sW), 默认为 **1**  
**padding** - 在输入的两端上的隐含零填充。可以是单个数字或者元组 (padT, padH, padW)。默认值: **0**  
**output\_padding**-在两端的进行**0**填充 ( $0 \leq \text{padding} < \text{stride}$ )，可以是单一数字或者元组 (out\_padT, out\_padH, out\_padW)。默认是 **0**。  
**dilation** - 内核元素之间的间距。可以是单个数字或者元组 (dT, dH, dW)。默认值: **1**  
**groups** - 将输入分成组, in\_channels应该被组数整除。默认值: **1**

举例：

```
>>> inputs = torch.randn(20, 16, 50, 10, 20)
>>> weights = torch.randn(16, 33, 3, 3, 3)
>>> F.conv_transpose3d(inputs, weights)
```

## 池化函数

```
torch.nn.functional.avg_pool1d(input, kernel_size, stride=None,
padding=0, ceil_mode=False, count_include_pad=True)
```

### source

对由几个输入平面组成的输入进行1D平均池化。有关详细信息和输出形状，参考[AvgPool1d](#)

参数：

```
input - 输入的张量 (minibatch x in_channels x iW)
kernel_size - 池化区域的大小，可以是单个数字或者元组 (kW)
stride - 池化操作的步长，可以是单个数字或者元组 (ssw)。默认值等于内核大小
padding - 在输入上隐式的零填充，可以是单个数字或者一个元组 (padw)，默认：0
ceil_mode - 当为True时，公式中将使用ceil而不是floor来计算输出形状。默认值：False
count_include_pad - 当为True时，将包括平均计算中的零填充。默认值：True
```

举例：

```
>>> # pool of square window of size=3, stride=2
>>> input = torch.tensor([[[[1, 2, 3, 4, 5, 6, 7]]]])
>>> F.avg_pool1d(input, kernel_size=3, stride=2)
tensor([[[2., 4., 6.]])
```

```
torch.nn.functional.avg_pool2d(input, kernel_size, stride=None,
padding=0, ceil_mode=False, count_include_pad=True) → Tensor
```

### source

通过步长 $dh \times dw$ 步骤在 $kh \times kw$ 区域中应用二维平均池操作。输出特征的数量等于输入平面的数量。

有关详细信息和输出形状，参考[AvgPool2d](#)

参数：

```
input - 输入的张量 (minibatch x in_channels x iH x iW)
kernel_size - 池化区域的大小，可以是单个数字或者元组 (kh x kw)
stride - 池化操作的步长，可以是单个数字或者元组 (sh x sw)。默认值等于内核大小
padding - 在输入上隐式的零填充，可以是单个数字或者一个元组 (padh x padw)，默认：0
ceil_mode - 当为True时，公式中将使用ceil而不是floor来计算输出形状。默认值：False
count_include_pad - 当为True时，将包括平均计算中的零填充。默认值：True
```

```
torch.nn.functional.avg_pool3d(input, kernel_size, stride=None, padding=0, ceil_mode=False, count_include_pad=True) → Tensor
```

## source

通过步长 $dt \times dh \times dw$ 步骤在 $kt \times kh \times kw$ 区域中应用3D平均池操作。输出特征的数量等于输入平面数/  $dt$ 。有关详细信息和输出形状，参考[AvgPool3d](#)

参数：

```
input - 输入的张量 (minibatch x in_channels x iT x iH x iW)
kernel_size - 池化区域的大小，可以是单个数字或者元组 (kT x kh x kw)
stride - 池化操作的步长，可以是单个数字或者元组 (sT x sh x sw)。默认值等于内核大小
padding - 在输入上隐式的零填充，可以是单个数字或者一个元组 (padT x padh x padw)，默认：0
ceil_mode - 当为True时，公式中将使用ceil而不是floor来计算输出形状。默认值：False
count_include_pad - 当为True时，将包括平均计算中的零填充。默认值：True
```

```
torch.nn.functional.max_pool1d(input, kernel_size, stride=None, padding=0, dilation=1, ceil_mode=False, return_indices=False)
```

### source

对由几个输入平面组成的输入进行1D最大池化。有关详细信息和输出形状，参考[MaxPool1d](#)

---

```
torch.nn.functional.max_pool2d(input, kernel_size, stride=None,
padding=0, dilation=1, ceil_mode=False, return_indices=False)
```

### source

对由几个输入平面组成的输入进行2D最大池化。有关详细信息和输出形状，参考[MaxPool2d](#)

---

```
torch.nn.functional.max_pool3d(input, kernel_size, stride=None,
padding=0, dilation=1, ceil_mode=False, return_indices=False)
```

### source

对由几个输入平面组成的输入进行3D最大池化。有关详细信息和输出形状，参考[MaxPool3d](#)

---

```
torch.nn.functional.max_unpool1d(input, indices, kernel_size, st
ride=None, padding=0, output_size=None)
```

计算MaxPool1d的部分逆。

有关详细信息和输出形状，参考[MaxUnPool1d](#)

---

```
torch.nn.functional.max_unpool2d(input, indices, kernel_size, st
ride=None, padding=0, output_size=None)
```

计算MaxPool2d的部分逆。

有关详细信息和输出形状，参考[MaxUnPool2d](#)

---

```
torch.nn.functional.max_unpool3d(input, indices, kernel_size, stride=None, padding=0, output_size=None)
```

计算MaxPool3d的部分逆。

有关详细信息和输出形状，参考[MaxUnPool3d](#)

---

```
torch.nn.functional.lp_pool1d(input, norm_type, kernel_size, stride=None, ceil_mode=False)
```

适用在几个输入平面组成的输入信号的1D power-平均池。

有关详细信息[LPPool1d](#)

---

```
torch.nn.functional.lp_pool2d(input, norm_type, kernel_size, stride=None, ceil_mode=False)
```

适用在几个输入平面组成的输入信号的2D power-平均池。

有关详细信息[LPPool2d](#)

---

```
torch.nn.functional.adaptive_max_pool1d(input, output_size, return_indices=False)
```

#### source

对由多个输入平面组成的输入进行1D自适应最大池化。

有关详细信息可见[AdaptiveMaxPool1d](#)

---

```
torch.nn.functional.adaptive_max_pool2d(input, output_size, return_indices=False)
```

#### source

对由多个输入平面组成的输入进行2D自适应最大池化。

---



有关详细信息可见[AdaptiveMaxPool2d](#)

---

```
torch.nn.functional.adaptive_max_pool3d(input, output_size, return_indices=False)
```

#### source

对由多个输入平面组成的输入进行3D自适应最大池化。

有关详细信息可见[AdaptiveMaxPool3d](#)

---

```
torch.nn.functional.adaptive_avg_pool1d(input, output_size) → Tensor
```

#### source

对由多个输入平面组成的输入进行1D自适应平均池化。

有关详细信息可见[AdaptiveAvgPool1d](#)

---

```
torch.nn.functional.adaptive_avg_pool2d(input, output_size) → Tensor
```

#### source

对由多个输入平面组成的输入进行2D自适应平均池化。

有关详细信息可见[AdaptiveAvgPool2d](#)

---

```
torch.nn.functional.adaptive_avg_pool3d(input, output_size) → Tensor
```

#### source

对由多个输入平面组成的输入进行3D自适应平均池化。

有关详细信息可见[AdaptiveAvgPool3d](#)

---

## 非线性激活函数

```
torch.nn.functional.threshold(input, threshold, value, inplace=False)
```

### source

对于输入的张量进行筛选

详细请看[Threshold](#)

---

```
torch.nn.functional.threshold_(input, threshold, value)
```

### source

和threshold函数一样

详细请看[Threshold](#)

---

```
torch.nn.functional.relu(input, inplace=False) → Tensor
torch.nn.functional.relu_(input) → Tensor
```

### source

对输入元素应用relu函数

详细请看[Relu](#)

---

```
torch.nn.functional.hardtanh(input, min_val=-1., max_val=1., inplace=False) → Tensor
torch.nn.functional.hardtanh_(input, min_val=-1., max_val=1.) → Tensor
```

### source

对输入元素应用HardTanh函数

详细请看[Hardtanh](#)

---

```
torch.nn.functional.relu6(input, inplace=False) → Tensor
```

#### source

对输入元素应用ReLU6函数  $\text{ReLU6}(x) = \min(\max(0, x), 6)$

详细请看[ReLU6](#)

---

```
torch.nn.functional.elu(input, alpha=1.0, inplace=False)
torch.nn.functional.elu_(input, alpha=1.) → Tensor
```

#### source

对输入元素应用ELU函数  $\text{ELU}(x) = \max(0, x) + \min(0, \alpha * (\exp(x) - 1))$

详细请看[ELU](#)

---

```
torch.nn.functional.selu(input, inplace=False) → Tensor[source]
```

详细可见[selu](#)

---

```
torch.nn.functional.leaky_relu(input, negative_slope=0.01, inplace=False) → Tensor
torch.nn.functional.leaky_relu_(input, negative_slope=0.01) → Tensor
```

详细可见[LeakyReLU](#)

---

```
torch.nn.functional.prelu(input, weight) → Tensor
```

详细可见[PReLU](#)

---

```
torch.nn.functional.rrelu(input, lower=1./8, upper=1./3, training=False, inplace=False) → Tensor
torch.nn.functional.rrelu_(input, lower=1./8, upper=1./3, training=False) → Tensor
```

详细可见[RRelu](#)

---

```
torch.nn.functional.glu(input, dim=-1) → Tensor
```

门控制线性单元  $H=A \times \sigma(B)$ , 输入张量将被按照特定维度分成一半是A，一半是B  
可以参看论文[Language Modeling with Gated Convolutional Networks](#)

参数：

```
input: 输入的张量
dim: 需要被分割的输入张量的维度
```

---

```
torch.nn.functional.logsigmoid(input) → Tensor
```

具体细节可以看[LogSigmoid](#)

---

```
torch.nn.functional.hardshrink(input, lambd=0.5) → Tensor
```

具体细节可以看[Hardshrink](#)

---

```
torch.nn.functional.tanhshrink(input, lambd=0.5) → Tensor
```

具体细节可以看[Tanhshrink](#)

---

```
torch.nn.functional.softsign(input, lambd=0.5) → Tensor
```

具体细节可以看[Softsign](#)

```
torch.nn.functional.softplus(input, beta=1, threshold=20) → Tensor
```

```
torch.nn.functional.softmin(input, dim=None, _stacklevel=3)
```

应用`softmin`函数 请注意 $\text{Softmin}(x) = \text{Softmax}(-x)$ 。请参阅数学公式的`softmax`定义。

具体细节可以看[Softmin](#)

参数：

`input`: 输入张量  
`dim`: `softmin`将被计算的维度（因此每个沿着`dim`的切分将总计为1）。

```
torch.nn.functional.softmax(input, dim=None, _stacklevel=3)
```

$\text{Softmax}(x_i) = \exp(x_i) / \sum_j \exp(x_j)$

它被应用于沿着对应维度的所有切分，并且将对它们进行重新缩放，以使得这些元素位于范围（0,1）中并且总计为1。

具体细节可以看[Softmax](#)

参数：

`input`: 输入张量  
`dim`: `softmax`被计算的维度。

```
torch.nn.functional.softshrink(input, lambd=0.5) → Tensor
```

具体细节可以看[Softshrink](#)

---

```
torch.nn.functional.log_softmax(input, dim=None, _stacklevel=3)
```

尽管在数学上等同于 $\log(\text{softmax}(x))$ ，但单独执行这两个操作会更慢，并且数值不稳定。该函数使用替代公式来正确计算输出和梯度。

具体细节可以看[LogSoftmax](#)

---

```
torch.nn.functional.tanh(input) → Tensor
```

具体细节可以看[Tanh](#)

---

```
torch.nn.functional.sigmoid(input) → Tensor
```

$\text{sigmoid}(x) = 1/(1+\exp(-x))$

具体细节可以看[Sigmoid](#)

## 归一化函数

---

```
torch.nn.functional.batch_norm(input, running_mean, running_var,
 weight=None, bias=None, training=False, momentum=0.1, eps=1e-05
)
```

[source](#)

具体细节可以看[BatchNorm1d](#) [BatchNorm2d](#) [BatchNorm3d](#)

---

```
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)
```

### source

具体细节可以看[InstanceNorm1d](#) [InstanceNorm2d](#) [InstanceNorm3d](#)

```
torch.nn.functional.layer_norm(input, normalized_shape, weight=None, bias=None, eps=1e-05)
```

具体细节可以看[LayerNorm](#)

```
torch.nn.functional.local_response_norm(input, size, alpha=0.0001, beta=0.75, k=1)
```

对由多个输入平面组成的输入应用本地响应规范化，其中通道占据第二维。通道应用归一化。

具体细节可以看[LocalResponseNorm](#)

```
torch.nn.functional.normalize(input, p=2, dim=1, eps=1e-12)
```

### source

对指定维度的输入执行 $L_p$ 标准化。

$$v = \frac{v}{\max(\|v\|_p, \epsilon)}$$

对于输入的维度 $\text{dim}$ 的每个子扩展 $v$ 。每个子扩张被平化成一个向量，即 $\|v\|_p$ 不是一个矩阵范数。

使用默认参数在第二维上用欧几里得范数进行归一化。

参数：

`input` - 输入张量的形状  
`p (float)` - 规范公式中的指数值。默认值：`2`  
`dim (int)` - 要缩小的维度。默认值：`1`  
`eps (float)` - 小值以避免除以零。默认值：`1e-12`

---

## 线性函数

```
torch.nn.functional.linear(input, weight, bias=None)
```

对于输入数据进行线性变化:  $y = xA^T + b$ .

形状：

```
Input: (N, *, in_features)(N, *, in_features) 这里的*表示为任意数量的附加
维度
Weight: (out_features, in_features)(out_features, in_features)
Bias: (out_features)(out_features)
Output: (N, *, out_features)
```

---

## Dropout 函数

```
torch.nn.functional.dropout(input, p=0.5, training=False, inplace=False)
```

[source](#)

```
torch.nn.functional.alpha_dropout(input, p=0.5, training=False)
```

[source](#)

详细可见[alpha\\_dropout](#)

参数：



`p(float, optional)` - 丢弃的可能性，默认是`0.5`  
`training(bool, optional)` - 在训练模型和验证模型之间的切换，默认是`false`

```
torch.nn.functional.dropout2d(input, p=0.5, training=False, inplace=False)
```

[source](#)

---

```
torch.nn.functional.dropout3d(input, p=0.5, training=False, inplace=False)
```

[source](#)

---

## 距离函数

```
torch.nn.functional.pairwise_distance(x1, x2, p=2, eps=1e-06, keepdim=False)
```

详细可见 [PairwiseDistance](#)

---

```
torch.nn.functional.cosine_similarity(x1, x2, dim=1, eps=1e-08)
```

[source](#)

计算向量 $v_1$ 、 $v_2$ 之间的距离 
$$\text{similarity} = \frac{x_1 \cdot x_2}{\max(\|v_1\|, \|v_2\|, \epsilon)}$$

参数：

x1 (Variable) - 首先输入参数。  
 x2 (Variable) - 第二个输入参数 (of size matching x1).  
 dim (int, optional) - 向量维数。默认为：1  
 eps (float, optional) - 小值避免被零分割。默认为：1e-8 模型：

形状：

input: (\*1,D,\*2)(\*1,D,\*2) D是位置维度  
 output: (\*1,\*2)(\*1,\*2) 1是位置维度

举例：

```
>>> input1 = autograd.Variable(torch.randn(100, 128))
>>> input2 = autograd.Variable(torch.randn(100, 128))
>>> output = F.cosine_similarity(input1, input2)
>>> print(output)
```

## 损失函数

torch.nn.functional.binary\_cross\_entropy(input, target, weight=None, size\_average=True, reduce=True)

### source

测量目标和输出之间的二进制交叉熵的函数。

详细可见[BCELoss](#)

参数：

input:任意维度  
 target:与输入维度相同  
 weight (张量, 可选): 如果提供的权重矩阵能匹配输入张量形状, 则手动调整重量  
 size\_average (布尔值, 可选): 默认情况下, 对每个小批次的损失进行平均观察。  
 但是, 如果field size\_average设置为False, 则每个小批次的损失将相加。 默认值: True  
 reduce (布尔值, 可选): 默认情况下, 根据size\_average的不同, 对每个小批次的损失进行平均或累计。 当reduce为False时, 将返回每个输入/目标元素的损失, 而忽略size\_average。 默认值: True

举例：

```
>>> 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()
```

```
torch.nn.functional.poisson_nll_loss(input, target, log_input=True,
full=False, size_average=True, eps=1e-08, reduce=True)
```

## source

泊松负对数似然损失 详细可见[PoissonNLLLoss](#)

参数：

input：按照泊松分布的期望  
target：随机样例  $\text{target} \sim \text{Poisson}(\text{input})$   
log\_input：如果为真，则损失计算为  $\exp(\text{input}) - \text{target} * \text{input}$ ，如果为False，则  $\text{input} - \text{target} * \log(\text{input} + \text{eps})$ 。默认值：True  
full：是否计算完全损失。即添加斯特林近似项。默认值：False  $\text{target} * \log(\text{target}) - \text{target} + 0.5 * \log(2 * \pi * \text{target})$   
size\_average：默认情况下，对每个小批次的损失进行平均观察。但是，如果field size\_average设置为False，则每个小批次的损失将相加。默认值：True  
eps (float, 可选) - 当log\_input="False"时避免评估  $\log(0)$   $\log(0)$  的较小值。默认：1e-8  
reduce (布尔值, 可选)：默认情况下，根据每个小批次的观测结果对损失进行平均，或根据size\_average进行汇总。如果reduce为False，则返回每批损失，并忽略size\_average。默认值：True

```
torch.nn.functional.cosine_embedding_loss(input1, input2, target,
margin=0, size_average=True, reduce=True) → Tensor
```

## source

详细可见[CosineEmbeddingLoss](#)

```
torch.nn.functional.cross_entropy(input, target, weight=None, size_average=True, ignore_index=-100, reduce=True)
```

## source

该标准将`log_softmax`和`nll_loss`结合在一个函数中。

## CrossEntropyLoss

参数：

`input` (张量) - (N,C) 其中，C 是类别的个数  
`target` (张量) - (N) 其大小是  $0 \leq \text{targets}[i] \leq C-1$ 。`weight` (Variable, optional) - (N) 其大小是  $0 \leq \text{targets}[i] \leq C-1$   
`weight` (张量, optional) - 为每个类别提供的手动权重。如果给出，必须是大小为C的张量  
`size_average` (bool, optional) - 默认情况下，是mini-batchloss的平均值；如果`size_average=False`，则是mini-batchloss的总和。  
`ignore_index` (int, 可选) - 指定被忽略且不对输入渐变有贡献的目标值。当`size_average`为`True`时，对非忽略目标的损失是平均的。默认值：`-100`  
`reduce` (布尔值, 可选) - 默认情况下，根据每个小批次的观测结果对损失进行平均，或根据`size_average`进行汇总。如果`reduce`为`False`，则返回每批损失，并忽略`size_average`。默认值：`True`

举例：

```
>>> input = torch.randn(3, 5, requires_grad=True)
>>> target = torch.randint(5, (3,), dtype=torch.int64)
>>> loss = F.cross_entropy(input, target)
>>> loss.backward()
```

```
torch.nn.functional.hinge_embedding_loss(input, target, margin=1.0, size_average=True, reduce=True) → Tensor
```

## source

详细可见[HingeEmbeddingLoss](#)

```
torch.nn.functional.kl_div(input, target, size_average=True) → Tensor
```

[source](#)

The [Kullback-Leibler divergence Loss](#)

详见[KLDivLoss](#)

参数：

```
input - 变量的任意形状
target - 与输入相同形状的变量
size_average - 如果是真的，输出就除以输入张量中的元素个数
reduce (布尔值，可选) - 默认情况下，根据每个小批次的观测结果对损失进行平均，
或根据size_average进行汇总。 如果reduce为False，则返回每批损失，并忽略size_average。 默认值：True
```

---

```
torch.nn.functional.l1_loss(input, target, size_average=True, reduce=True) → Tensor
```

详见[L1Loss](#)

---

```
torch.nn.functional.mse_loss(input, target, size_average=True, reduce=True) → Tensor
```

详见[MSELoss](#)

---

```
torch.nn.functional.margin_ranking_loss(input, target, size_average=True, reduce=True) → Tensor
```

详见[MarginRankingLoss](#)

---

```
torch.nn.functional.multilabel_soft_margin_loss(input, target, size_average=True, reduce=True) → Tensor
```

详细可见[MultiLabelSoftMarginLoss](#)

```
torch.nn.functional.multi_margin_loss(input, target, p=1, margin=1, weight=None, size_average=True, reduce=True) → Tensor
```

详细可见[MultiMarginLoss](#)

```
torch.nn.functional.nll_loss(input, target, weight=None, size_average=True, ignore_index=-100, reduce=True)
```

详细可见[NLLLoss](#)

参数：

`input` -  $\backslash (N, C) \backslash$  其中  $C$  = 类的数量或 (  $2D, Loss$  ) 的情况下的  $(N, C, H, W)$  `target` -  $\backslash (N) \backslash$  , 其中每个值为  $0 \leq targets[i] \leq C-1$   
`weight` (可变, 可选) - 给每个类别的手动重新调整重量。如果给定, 必须变量大小是  $C$   
`size_average` (bool, 可选) - 默认情况下, 损失是对每个小型服务器的观察值进行平均。如果 `size_average` 为 `False`, 则对于每个 minibatch 都会将损失相加。默认值: `True`  
`ignore_index` (int, 可选) - 指定被忽略且不对输入渐变有贡献的目标值。当 `size_average` 为 `True` 时, 对非忽略目标的损失是平均的。默认值: `-100`

举例：

```
>>> # 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()
```

```
torch.nn.functional.binary_cross_entropy_with_logits(input, target, weight=None, size_average=True, reduce=True)
```

## source

测量目标和输出逻辑之间二进制十进制熵的函数

详细可见[BCEWithLogitsLoss](#)

参数：

```
input - 任意形状 of 变量
target - 与输入形状相同的变量
weight (可变, 可选) - 手动重量, 如果提供重量以匹配输入张量形状
size_average (bool, 可选) - 默认情况下, 损失是对每个小型服务器的观察值进行平均。然而, 如果字段sizeAverage设置为False, 则相应的损失代替每个minibatch的求和。默认值: True
reduce (布尔值, 可选) - 默认情况下, 根据每个小批次的观测结果对损失进行平均, 或根据size_average进行汇总。 如果reduce为False, 则返回每批损失, 并忽略size_average。 默认值: True
```

举例：

```
>>> input = torch.randn(3, requires_grad=True)
>>> target = torch.empty(3).random_(2)
>>> loss = F.binary_cross_entropy_with_logits(input, target)
>>> loss.backward()
```

```
torch.nn.functional.smooth_l1_loss(input, target, size_average=True, reduce=True) → Tensor
```

详细可见[SmoothL1Loss](#)

```
torch.nn.functional.soft_margin_loss(input, target, size_average=True, reduce=True) → Tensor
```

详细可见[Soft\\_margin\\_loss](#)

```
torch.nn.functional.triplet_margin_loss(anchor, positive, negative, margin=1.0, p=2, eps=1e-06, swap=False, size_average=True, reduce=True)
```

详细可见[TripletMarginLoss](#)

## 视觉函数

```
torch.nn.functional.pixel_shuffle(input, upscale_factor)
```

### source

将形状为 $[*, C_r^2, H, W]$ 的张量重新排列成形状为 $[C, H_r, W_r]$ 的张量.

详细请看[PixelShuffle](#)

参数：

```
input (Variable) - 输入
upscale_factor (int) - 增加空间分辨率的因子。
```

举例：

```
>>> ps = nn.PixelShuffle(3)
>>> input = autograd.Variable(torch.Tensor(1, 9, 4, 4))
>>> output = ps(input)
>>> print(output.size())
torch.Size([1, 1, 12, 12])
```

```
torch.nn.functional.pad(input, pad, mode='constant', value=0)
```

### source

填充张量.

可以参考[torch.nn.ConstantPad2d](#), [torch.nn.ReflectionPad2d](#), 和 [torch.nn.ReplicationPad2d](#)对于每个填充模式如何工作的具体例子。



参数：

```
input (Variable) - 4D 或 5D tensor
pad (tuple) - 4元素 或 6-元素 tuple
mode - 'constant', 'reflect' or 'replicate'
value - 用于constant padding 的值.
```

举例：

```
>>> 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)
>>> 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])
```

```
torch.nn.functional.upsample(input, size=None, scale_factor=None,
 mode='nearest', align_corners=None)
```

[source](#)

Upsamples输入内容要么就是给定的size或者scale\_factor 用于采样的算法是由模型决定的 目前支持的是空间和容量的采样，即期望输入的形状是4-d或5-d。输入尺寸被解释为:迷你批x通道x深度x高度x宽度 用于upsampling的模型是:线性的(仅3D)，双线性的(仅4D)，三线性(仅5D)

参数：

`input (Variable)` - 输入内容  
`size (int or Tuple[int, int] or Tuple[int, int, int])` - 输出空间的大小。  
`scale_factor (int)` - 乘数的空间大小。必须是一个整数。  
`mode (string)` - 用于向上采样的算法: 'nearest' | 'bilinear' | 'trilinear'  
`align_corners (bool, 可选)` - 如果为`True`，则输入和输出张量的角点像素对齐，从而保留这些像素的值。这只在模式为线性，双线性或三线性时才有效。默认值：`False`

### 警告：

使用`align_corners = True`时，线性插值模式（线性，双线性和三线性）不会按比例对齐输出和输入像素，因此输出值可能取决于输入大小。  
 这是这些模式到版本`0.3.1`的默认行为。此后，默认行为是`align_corners = False`。  
 有关这将如何影响输出的具体示例，请参见`Upsample`。

```
torch.nn.functional.upsample_nearest(input, size=None, scale_factor=None)
```

**source** 使用最接近的邻居的像素值来对输入进行采样。

### 警告：

此功能已弃用，以支持`torch.nn.functional.upsample()`。这相当于`nn.functional.upsample(..., mode='nearest')`。

目前支持空间和体积上采样（即预期的输入是4或5维）。

### 参数：

`input (Variable)` - 输入内容  
`size (int or Tuple[int, int])` - 输出空间的大小。  
`scale_factor (int or Tuple[int, int])` - 乘数的空间大小

```
torch.nn.functional.upsample_bilinear(input, size=None, scale_factor=None)
```

[source](#) 使用双线性向上采样来扩展输入

警告：

这个函数是被弃用的。使用`nn.functional.upsample`相反 预期的输入是空间的(4维)。  
使用`upsample_trilinear`来进行体积(5维)输入。

参数：

`input` (Variable) - 输入内容  
`size` (int or Tuple[int, int]) - 输出空间的大小。  
`scale_factor` (int or Tuple[int, int]) - 乘数的空间大小

```
torch.nn.functional.grid_sample(input, grid, mode='bilinear', padding_mode='zeros')
```

[source](#)

给定输入和流场网格，使用网格中的输入像素位置计算输出。

使用双线性插值来对输入像素进行采样。目前，仅支持空间（4维）和体积（5维）输入。

对于每个输出位置，网格具有用于计算输出的`x`，`y`输入像素位置。在5D输入的情况下，网格具有`x`，`y`，`z`像素位置。

`grid`的值在`[-1, 1]`的范围内。这是因为像素位置是由输入高度和宽度标准化的。

例如，值：`x: -1`，`y: -1`是输入的左上像素，值：`x: 1`，`y: 1`是输入的右下像素。

如果`grid`的值超出`[-1, 1]`的范围，则按`padding_mode`的定义处理这些位置。选项为零或边框，定义这些位置以使用0或图像边界值作为对双线性插值的贡献。

参数：

```

input (Tensor) - 输入批次 (N x C x IH x IW) 或 (N x C x ID x IH x IW)
grid (Tensor) - 尺寸 (N x OH x OW x 2) 或 (N x OD x OH x OW x 3) 的流场，
padding_mode (str) - 用于外部网格值“零”的填充模式 '边境'。默认值：'零'

```

返回：

```
output (tensor)
```

```
torch.nn.functional.affine_grid(theta, size)
```

#### source

生成一个2d流场，给定一批仿射矩阵`theta`通常与`grid_sample()`一起使用来实现 Spatial Transformer Networks。

参数：

```

theta (Tensor) - 输入一批仿射矩阵 (N×2×3N×2×3)
size (torch.Size) - 目标输出图像尺寸 (N×C×H×W×C×H×W) 例如：torch.Size((32, 3, 24, 24))

```

返回：

```
output (tensor) : 输出张量大小 (N×H×W×2N×H×W×2)
```

## 并行函数（多GPU，分布式）

```
torch.nn.parallel.data_parallel(module, inputs, device_ids=None,
 output_device=None, dim=0, module_kwargs=None)
```

#### source

在device\_ids中给出的GPU上并行评估模块（输入）。这是DataParallel模块的功能版本。

参数：

```
module - 并行评估的模块
input - 输入到模块
device_ids - 要在其上复制模块的GPU ID
output_device - 输出的GPU位置使用-1指示CPU。（默认：device_ids [0]）
```

返回：

包含位于output\_device上的模块（输入）结果的张量

译者署名

用户名	头像	职能	签名
travel		翻译	人生总 要追求 点什么

## 自动差异化包 - torch.autograd

- [Variable](#)
- [API 兼容性](#)
- [变量的直接操作](#)
- [In-place 正确性检查](#)
- [Function](#)

`torch.autograd` 提供了类和函数用来对任意标量函数进行求导。要想使用自动求导，只需要对已有的代码进行微小的改变。只需要将所有的 `tensor` 包含进 `Variable` 对象中即可。

### `torch.autograd.backward(variables, grad_variables, retain_variables=False)`

计算给定变量wrt图叶的梯度的总和。该图使用链规则进行区分。如果任何 `variables` 非标量（即它们的数据具有多个元素）并且需要梯度，则该函数另外需要指定 `grad_variables`。它应该是一个匹配长度的序列，其包含差分函数wrt对应变量的渐变（`None`对于不需要梯度张量的所有变量，它是可接受的值）。

此函数在树叶中累加梯度 - 在调用它之前可能需要将其清零。

参数:

- `variables` (variable 列表) – 将计算导数的变量。
- `grad_variables` (序列( `Tensor` , `Variable` 或者 `None` )) – 渐变写入相应变量的每个元素。任何张量将被自动转换为`volatile`，除非 `create_graph`为`True`。可以为标量变量或不需要`grad`的值指定无值。如果所有`grad_variables`都可以接受`None`值，则该参数是可选的。
- `retain_graph` (bool, 可选) - 如果为`False`，则用于计算`grad`的图形将被释放。请注意，在几乎所有情况下，将此选项设置为`True`不是必需的，通常可以以更有效的方式解决。默认值为`create_graph`。
- `create_graph` (bool, 可选) - 如果为`true`，则构造导数的图形，允许计算更高阶的衍生产品。默认为`False`，除非`grad_variables`包含至少一个非易失性变量。

### `torch.autograd.grad(outputs, inputs, grad_outputs=None, retain_graph=None, create_graph=None, only_inputs=True)`

计算并返回输入的输出梯度的总和。

**gradoutputs**应该是**output** 包含每个输出的预先计算的梯度的长度匹配序列。如果输出不需要 **grad**，则渐变可以是**None**）。当不需要派生图的图形时，梯度可以作为**Tensors**给出，或者作为**Variables**，在这种情况下将创建图形。

如果**only\_inputs**为**True**，该函数将仅返回指定输入的渐变列表。如果它是**False**，则仍然计算所有剩余叶子的渐变度，并将累积到其**grad** 属性中。

参数：

- **outputs**（可变序列） - 差分函数的输出。
- **inputs**（可变序列） - 输入将返回梯度的积分（并不积累**grad**）。
- **grad\_outputs**（**Tensor** 或**Variable**的序列） - 渐变wrt每个输出。任何张量将被自动转换为**volatile**，除非**create\_graph**为**True**。可以为标量变量或不需要**grad**的值指定无值。如果所有**grad\_variables**都可以接受**None**值，则该参数是可选的。
- **retain\_graph**（**bool**，可选） - 如果为**False**，则用于计算**grad**的图形将被释放。请注意，在几乎所有情况下，将此选项设置为**True**不是必需的，通常可以以更有效的方式解决。默认值为**create\_graph**。
- **create\_graph**（**bool**，可选） - 如果为**True**，则构造导数的图形，允许计算高阶衍生产品。默认为**False**，除非**grad\_variables**包含至少一个非易失性变量。
- **only\_inputs**（**bool**，可选） - 如果为**True**，则渐变wrt离开是图形的一部分，但不显示**inputs**不会被计算和累积。默认为**True**。

## Variable

### API 兼容性

可变API与常规**Tensor** API（除了几个就地方法，覆盖渐变计算所需的输入之外）几乎相同。在大多数情况下，传感器可以安全地替换为变量，代码将保持工作正常。因此，我们不会记录变量的所有操作，**torch.Tensor**为此也应该参考文档。

### 变量的直接操作

支持自动归档中的就地操作是一件很困难的事情，我们在大多数情况下都不鼓励使用它们。**Autograd**的积极缓冲区释放和重用使其非常高效，并且在现场操作实际上会降低内存使用量的情况下，极少数场合很少。除非您在内存压力很大的情况下运行，否则您可能永远不需要使用它们。

### In-place 正确性检查

所有**Variable**的跟踪应用于它们的就地操作，并且如果实现检测到在一个功能中保存了向后的变量，但是之后被修改就位，则一旦向后通过开始就会产生错误。这可以确保如果您使用就地功能并且没有看到任何错误，则可以确定计算出的渐变是正确的。

### class torch.autograd.Variable

包裹张量并记录应用的操作。

变量是Tensor对象周围的薄包装，它也保存了渐变wrt，并引用了创建它的函数。此引用允许回溯创建数据的整个操作链。如果变量是由用户创建的，那么它的grad\_fn将被None我们称为这样的对象叶变量。

由于自动调整仅支持标量值函数微分，因此grad大小始终与数据大小匹配。此外，grad通常仅分配给叶变量，否则将始终为零。

变量：

- data – 包含的 Tensor
- grad - 变量保持类型和位置的坡度匹配.data。此属性是懒惰分配的，无法重新分配。
- requires\_grad - 指示变量是否由包含任何需要的变量的子图创建的布尔值。有关详细信息，请参阅从向后排除子图。只能在叶变量上更改。
- volatile - Boolean表示在推断模式下应该使用Variable，即不保存历史记录。有关详细信息，请参阅从向后排除子图。只能在叶变量上更改。
- is\_leaf - Boolean指示变量是否为图形叶（即如果由用户创建）。
- grad\_fn - 梯度函数图跟踪。

属性：

- data (any tensor class) – 被包含的 Tensor
- requires\_grad (bool) – requires\_grad 标记. 只能通过 keyword 传入.
- volatile (bool) – volatile 标记. 只能通过 keyword 传入.

## backward(**gradient=None, retain\_variables=False**)

当前 Variable 对 leaf variable 求偏导。

计算图可以通过链式法则求导。如果 Variable 是非标量( non-scalar )的，且 requires\_grad=True 。那么此函数需要指定 gradient ，它的形状应该和 Variable 的长度匹配，里面保存了 Variable 的梯度。

此函数累积 leaf variable 的梯度。你可能需要在调用此函数之前将 Variable 的梯度置零。

参数：

- grad\_variables (Tensor, Variable或None) - 渐变wrt变量。如果它是张量，它将被自动转换为volatile，除非create\_graph是True。可以为标量变量或不需要grad的值指定无值。如果无值可接受，则此参数是可选的。
- retain\_graph (bool, 可选) - 如果为False，用于计算grads的图形将被释放。请注意，在几乎所有情况下，将此选项设置为True不是必需的，通常可以以更有效的方式解决。默认值为 create\_graph。
- create\_graph (bool, 可选) - 如果为true，则构造导数的图形，允许计算更高



阶的衍生产品。默认为False，除非gradient是volatile变量。

## detach()

返回一个新变量，与当前图形分离。

结果将永远不需要渐变。如果输入是易失的，输出也将变得不稳定。

注意：返回变量使用与原始变量相同的数据张量，并且可以看到其中任何一个的就地修改，并且可能会触发正确性检查中的错误。

## detach\_()

从创建它的图形中分离变量，使其成为叶。

## register\_hook(hook)

注册一个 backward 钩子。

每次 gradients 被计算的时候，这个 hook 都被调用。hook 应该拥有以下签名：

```
hook(grad) -> Variable or None
```

钩子不应该修改它的参数，但是它可以选择返回一个新的渐变，用来代替它grad。

此函数返回一个带有handle.remove()从模块中删除钩子的方法的句柄。

例：

```
>>> v = Variable(torch.Tensor([0, 0, 0]), requires_grad=True)
>>> h = v.register_hook(lambda grad: grad * 2) # double the gra
dient
>>> v.backward(torch.Tensor([1, 1, 1]))
>>> v.grad.data
 2
 2
 2
[torch.FloatTensor of size 3]
>>> h.remove() # removes the hook
```

## reinforce(reward)

注册一个奖励，这个奖励是由一个随机过程得到的。

微分一个随机节点需要提供一个奖励值。如果你的计算图中包含随机 `operations`，你需要在他们的输出上调用这个函数。否则的话，会报错。

参数:

- `reward (Tensor)` – 每个元素的reward张量。必须和 `Variable` 形状相同，并在同一个设备上。

## `retain_grad()`

启用非叶变量的`.grad`属性。

## `class torch.autograd.Function`

记录操作历史并定义用于区分操作的公式。

在Variables上执行的每个操作创建一个新的函数对象，执行计算，并记录它发生的情况。历史以DAG的形式保留，边缘表示数据依赖（）。然后，当调用向后时，通过调用每个对象的方法，并将返回的梯度传递给下一个s来处理图形的拓扑排序。  
`input <- output.backward()`FunctionFunction

通常，用户与功能交互的唯一方式是创建子类并定义新的操作。这是延长`torch.autograd`的推荐方法。

由于函数逻辑是大多数脚本的热点，因此几乎所有脚本都被移动到C后端，以确保框架开销最小化。

每个功能仅用于一次（在正向通行证中）。

变量:

- `saved_tensors` - 保存在呼叫中的Tensor元组 `forward()`。
- `saved_variables` - 对应于调用中保存的张量的变量元组`forward()`。
- `needs_input_grad` - 长度的布尔元组`num_inputs`，指示给定输入是否需要渐变。这可以用于优化缓冲区保存为向后，忽略梯度计算`backward()`。
- `num_inputs` - 给出的输入数量`forward()`。
- `num_outputs` - 返回的张量数`forward()`。
- `requires_grad` - 布尔值，表示是否`backward()`需要调用。

## `backward(* grad_output)`

定义用于区分操作的公式。

此功能将被所有子类覆盖。

所有参数都是张量。它必须接受完全一样多的论据，因为许多输出都`forward()`返回，它应该返回尽可能多的张量，因为有输入`forward()`。每个参数是给定输出的渐变wrt，每个返回的值应该是对应输入的渐变。


**static forward(*args*, \**kwargs*)**

执行操作。

此功能将被所有子类覆盖。

它可以采取并返回任意数量的张量。

译者署名

用户名	头像	职能	签名
Song		翻译	人生总要追求点什么

# torch.optim

1. 如何使用optimizer
  - 构建
  - 为每个参数单独设置选项
  - 进行单次优化
  - optimizer.step ()
  - optimizer.step(closure))
2. 算法
3. 如何调整学习率

`torch.optim` 是实现各种优化算法的包。最常用的方法都已经支持，接口很常规，所以以后也可以很容易地集成更复杂的方法。

## 如何使用optimizer

要使用 `torch.optim`，您必须构造一个 `optimizer` 对象。这个对象能保存当前的参数状态并且基于计算梯度更新参数

### 构建

要构造一个 `Optimizer`，你必须给它一个包含参数（必须都是 `Variable` 对象）进行优化。然后，您可以指定 `optimizer` 的参数选项，比如学习率，权重衰减等。

例子：

```
optimizer = optim.SGD(model.parameters(), lr = 0.01, momentum=0.9)
optimizer = optim.Adam([var1, var2], lr = 0.0001)
```

### 为每个参数单独设置选项

`Optimizer` 也支持为每个参数单独设置选项。若想这么做，不要直接传入 `Variable` 的 `iterable`，而是传入 `dict` 的 `iterable`。每一个 `dict` 都分别定义了一组参数，并且包含一个 `param` 键，这个键对应参数的列表。其他的键应该 `optimizer` 所接受的其他参数的关键字相匹配，并且会被用于对这组参数的优化。

注意：

您仍然可以将选项作为关键字参数传递。它们将被用作默认值，在不覆盖它们的组中。当您只想改变一个选项，同时保持参数组之间的所有其他选项一致时，这很有用。

例如，当我们想指定每一层的学习率时，这是非常有用的：

```
optim.SGD([
 {'params': model.base.parameters()},
 {'params': model.classifier.parameters(), 'lr': 1e-3
}], lr=1e-2, momentum=0.9)
```

这意味着 `model.base` 参数将使用默认的学习速率 `1e-2`，`model.classifier` 参数将使用学习速率 `1e-3`，并且 `0.9` 的 `momentum` 将会被用于所有的参数。

## 进行单次优化

所有的 `optimizer` 都会实现 `step()` 更新参数的方法。它能按两种方式来使用：

### `optimizer.step()`

这是大多数 `optimizer` 所支持的简化版本。一旦梯度被如 `backward()` 之类的函数计算好后，我们就可以调用该函数。

例子

```
for input, target in dataset:
 optimizer.zero_grad()
 output = model(input)
 loss = loss_fn(output, target)
 loss.backward()
 optimizer.step()
```

### `optimizer.step(closure)`

一些优化算法例如 `Conjugate Gradient` 和 `LBFGS` 需要重复多次计算函数，因此你需要传入一个闭包去允许它们重新计算你的模型。这个闭包会清空梯度，计算损失，然后返回。

例子：

```

for input, target in dataset:
 def closure():
 optimizer.zero_grad()
 output = model(input)
 loss = loss_fn(output, target)
 loss.backward()
 return loss
 optimizer.step(closure)

```

## 算法

```
class torch.optim.Optimizer(params, defaults)
```

所有优化的基类。

参数：

1. **params (iterable)** —— 可迭代的 `Variable` 或者 `dict`。指定应优化哪些变量。
2. **defaults-(dict)**：包含优化选项的默认值的`dict`（一个参数组没有指定的参数选项将会使用默认值）。

```
load_state_dict(state_dict)
```

加载 `optimizer` 状态

参数：

1. **state\_dict (dict)** —— `optimizer` 的状态。应该是 `state_dict()` 调用返回的对象。

```
state_dict()
```

将优化器的状态返回为一个 `dict`。

它包含两个内容：

1. **state** - 持有当前 `optimization` 状态的 `dict`。它包含了 优化器类之间的不同。
2. **param\_groups** - 一个包含了所有参数组的 `dict`。

```
step(closure)
```

执行单个优化步骤（参数更新）。

参数：

1. `closure` (callable, 可选) – 重新评估模型并返回损失的闭包。 `hon zero_grad()`

清除所有优化过的 `Variable` 的梯度。

```
class torch.optim.Adadelta(params, lr=1.0, rho=0.9, eps=1e-06, weight_decay=0)
```

实现 `Adadelta` 算法。

[ADADELTA](#)中提出了一种[自适应学习速率法](#)。

参数：

1. `params` (iterable) – 用于优化的可以迭代参数或定义参数组
2. `rho` (float, 可选) – 用于计算平方梯度的运行平均值的系数（默认值：0.9）
3. `eps` (float, 可选) – 增加到分母中以提高数值稳定性的术语（默认值：1e-6）
4. `lr` (float, 可选) – 将`delta`应用于参数之前缩放的系数（默认值：1.0）
5. `weight_decay` (float, 可选) – 权重衰减 (L2范数)（默认值: 0）

```
step(closure)
```

执行单个优化步骤。

参数：

1. `closure` (callable, 可选) – 重新评估模型并返回损失的闭包。

```
class torch.optim.Adagrad(params, lr=0.01, lr_decay=0, weight_decay=0)
```

实现`Adagrad`算法。

在[在线学习和随机优化的自适应子梯度方法](#)中被提出。

参数：

1. `params` (iterable) – 用于优化的可以迭代参数或定义参数组
2. `lr` (float, 可选) – 学习率（默认: 1e-2）
3. `lr_decay` (float, 可选) – 学习率衰减（默认: 0）
4. `weight_decay` (float, 可选) – 权重衰减（L2范数）（默认: 0）

```
step(closure)
```

执行单个优化步骤。

参数：

1. `closure` (callable, 可选) – 重新评估模型并返回损失的闭包。

```
class torch.optim.Adam(params, lr=0.001, betas=(0.9, 0.999), eps=1e-08, weight_decay=0)[source]
```

实现Adam算法。

它在[Adam: A Method for Stochastic Optimization](#)中被提出。

参数：

1. `params` (iterable) – 用于优化的可以迭代参数或定义参数组
2. `lr` (float, 可选) – 学习率（默认：1e-3）
3. `betas` (Tuple[float, float], 可选) – 用于计算梯度运行平均值及其平方的系数（默认：0.9，0.999）
4. `eps` (float, 可选) – 增加分母的数值以提高数值稳定性（默认：1e-8）
5. `weight_decay` (float, 可选) – 权重衰减（L2范数）（默认：0）

```
step(closure)
```

执行单个优化步骤。

参数：

1. `closure` (callable, 可选) – 重新评估模型并返回损失的闭包。

```
class torch.optim.Adamax(params, lr=0.002, betas=(0.9, 0.999), eps=1e-08, weight_decay=0)
```

实现 Adamax 算法（Adam的一种基于无穷范数的变种）。

它在[Adam: A Method for Stochastic Optimization](#)中被提出。

参数：

1. `params` (iterable) – 用于优化的可以迭代参数或定义参数组
2. `lr` (float, 可选) – 学习率（默认：2e-3）
3. `betas` (Tuple[float, float], 可选) – 用于计算梯度以及梯度平方的运行平均值的系数
4. `eps` (float, 可选) – 增加分母的数值以提高数值稳定性（默认：1e-8）
5. `weight_decay` (float, 可选) – 权重衰减（L2范数）（默认：0）

```
step(closure=None)
```

执行单个优化步骤。



参数：

1. `closure` (callable, 可选) – 重新评估模型并返回损失的闭包。

```
class torch.optim.ASGD(params, lr=0.01, lambd=0.0001, alpha=0.75,
 t0=1000000.0, weight_decay=0)
```

实现平均随机梯度下降。

它在 [Acceleration of stochastic approximation by averaging](#) 中被提出。

参数：

1. `params` (iterable) – 用于优化的可以迭代参数或定义参数组
2. `lr` (float, 可选) – 学习率（默认：1e-2）
3. `lambd` (float, 可选) – 衰减期（默认：1e-4）
4. `alpha` (float, 可选) – eta更新的指数（默认：0.75）
5. `t0` (float, 可选) – 指明在哪一次开始平均化（默认：1e6）
6. `weight_decay` (float, 可选) – 权重衰减（L2范数）（默认：0）

```
step(closure)
```

执行单个优化步骤。

参数：

1. `closure` (callable, 可选) – 重新评估模型并返回损失的闭包。

```
class torch.optim.LBFGS(params, lr=1, max_iter=20, max_eval=None,
 tolerance_grad=1e-05, tolerance_change=1e-09, history_size=100,
 line_search_fn=None)
```

实现L-BFGS算法。

警告: 这个optimizer不支持为每个参数单独设置选项以及不支持参数组（只能有一个）现在所有参数必须在单个设备上。将来会有所改善。

注意: 这是一个内存高度密集的optimizer（它要求额外的 `param_bytes * (history_size + 1)` 个字节）。如果它不适应内存，尝试减小history size，或者使用不同的算法。

参数：

1. `lr` (float) – 学习率（默认：1）
2. `max_iter` (int) – 每个优化步骤的最大迭代次数（默认：20）
3. `max_eval` (int) – 每个优化步骤的最大函数评估次数（默认：max \* 1.25）
4. `tolerance_grad` (float) – 一阶最优的终止容忍度（默认：1e-5）
5. `tolerance_change` (float) – 功能值/参数更改的终止公差（默认：1e-9）

## 6. history\_size (int) – 更新历史记录大小（默认：100）

```
step(closure)
```

执行单个优化步骤。

参数：

1. closure (callable, 可选) – 重新评估模型并返回损失的闭包。

```
class torch.optim.RMSprop(params, lr=0.01, alpha=0.99, eps=1e-08,
weight_decay=0, momentum=0, centered=False)[source]
```

实现 RMSprop 算法。

由 G. Hinton 在他的[课程中提出](#)。

中心版本首次出现在[Generating Sequences With Recurrent Neural Networks](#)。

参数：

1. params (iterable) – 用于优化的可以迭代参数或定义参数组
2. lr (float, 可选) – 学习率（默认：1e-2）
3. momentum (float, 可选) – 动量因子（默认：0）
4. alpha (float, 可选) – 平滑常数（默认：0.99）
5. eps (float, 可选) – 增加分母的数值以提高数值稳定性（默认：1e-8）
6. centered (bool, 可选) – 如果为 True，计算中心化的 RMSProp，通过其方差的估计来对梯度进行归一化
7. weight\_decay (float, 可选) – 权重衰减（L2范数）（默认：0）

```
step(closure)
```

执行单个优化步骤。

参数：

1. closure (callable, 可选) – 重新评估模型并返回损失的闭包。

```
class torch.optim.Rprop(params, lr=0.01, etas=(0.5, 1.2), step_sizes=(1e-06, 50))
```

实现弹性反向传播算法。

参数：

1. params (iterable) – 用于优化的可以迭代参数或定义参数组
2. lr (float, 可选) – 学习率（默认：1e-2）

3. `etas` (Tuple[float, float], 可选) – 一对 (`etaminus`, `etaplis`)，它们是乘数增加和减少因子（默认：0.5，1.2）
4. `step_sizes` (Tuple[float, float], 可选) – 允许的一对最小和最大的步长（默认：1e-6，50）

```
step(closure)
```

执行单个优化步骤。

参数：

1. `closure` (callable, 可选) – 重新评估模型并返回损失的闭包。

```
class torch.optim.SGD(params, lr=, momentum=0, dampening=0, weight_decay=0, nesterov=False)
```

实现随机梯度下降算法（`momentum` 可选）。

Nesterov 动量基于 [On the importance of initialization and momentum in deep learning](#) 中的公式。

参数：

1. `params` (iterable) – 用于优化的可以迭代参数或定义参数组
2. `lr` (float) – 学习率
3. `momentum` (float, 可选) – 动量因子（默认：0）
4. `weight_decay` (float, 可选) – 权重衰减（L2 范数）（默认：0）
5. `dampening` (float, 可选) – 动量的抑制因子（默认：0）
6. `nesterov` (bool, 可选) – 使用 Nesterov 动量（默认：False）

例子：

```
>>> optimizer = torch.optim.SGD(model.parameters(), lr=0.1, momentum=0.9)
>>> optimizer.zero_grad()
>>> loss_fn(model(input), target).backward()
>>> optimizer.step()
```

提示：

带有动量/Nesterov的SGD的实现稍微不同于Sutskever等人以及其他框架中的实现。考虑到Momentum的具体情况，更新可以写成  $v = p * v + g$   $p = p - lr * v$  其中， $p$ 、 $g$ 、 $v$ 和 $p$ 分别是参数、梯度、速度和动量。这是在对比Sutskever et al。和其他框架采用该形式的更新  $v = p * v + lr * g$   $p = p - v$  Nesterov版本被类似地修改。

```
step(closure)
```

执行单个优化步骤。

参数：

1. `closure` (callable, 可选) – 重新评估模型并返回损失的闭包。

## 如何调整学习率

`torch.optim.lr_scheduler` 提供了几种方法来根据 `epoches` 的数量调整学习率。`torch.optim.lr_scheduler.ReduceLROnPlateau` 允许基于一些验证测量来降低动态学习速率。

```
class torch.optim.lr_scheduler.LambdaLR(optimizer, lr_lambda, last_epoch=-1)
```

将每个参数组的学习速率设置为初始的 `lr` 乘以一个给定的函数。当 `last_epoch=-1` 时，将初始 `lr` 设置为 `lr`。

参数：

1. `optimizer` (Optimizer) – 包装的优化器。
2. `lr_lambda` (function or list) – 一个函数来计算一个乘法因子给定一个整数参数的 `epoch`，或列表等功能，为每个组 `optimizer.param_groups`。
3. `last_epoch` (int) – 最后一个时期的索引。默认: -1.

例子：

```
>>> # Assuming optimizer has two groups.
>>> lambda1 = lambda epoch: epoch // 30
>>> lambda2 = lambda epoch: 0.95 ** epoch
>>> scheduler = LambdaLR(optimizer, lr_lambda=[lambda1, lambda2])
>>> for epoch in range(100):
>>> scheduler.step()
>>> train(...)
>>> validate(...)
```

```
class torch.optim.lr_scheduler.StepLR(optimizer, step_size, gamma=0.1, last_epoch=-1)
```

将每个参数组的学习速率设置为每个 `step_size` 时间段由 `gamma` 衰减的初始 `lr`。当 `last_epoch = -1` 时，将初始 `lr` 设置为 `lr`。

1. optimizer (Optimizer) – 包装的优化器。
2. step\_size (int) – 学习率衰减期。
3. gamma (float) – 学习率衰减的乘积因子。默认值:-0.1。
4. last\_epoch (int) – 最后一个时代的指数。默认值:1。

例子：

```
>>> # Assuming optimizer uses lr = 0.5 for all groups
>>> # lr = 0.05 if epoch < 30
>>> # lr = 0.005 if 30 <= epoch < 60
>>> # lr = 0.0005 if 60 <= epoch < 90
>>> # ...
>>> scheduler = StepLR(optimizer, step_size=30, gamma=0.1)
>>> for epoch in range(100):
>>> scheduler.step()
>>> train(...)
>>> validate(...)
```

```
class torch.optim.lr_scheduler.MultiStepLR(optimizer, milestones
, gamma=0.1, last_epoch=-1)
```

一旦时间的数量达到一个里程碑,则将每个参数组的学习率设置为伽玛衰减的初始值。当last\_epoch=-1时,将初始lr设置为lr。

参数：

1. optimizer (Optimizer) – 包装的优化器。
2. milestones (list) – 时期指标的列表。必须增加。
3. gamma (float) – 学习率衰减的乘积因子。默认: -0.1.
4. last\_epoch (int) – 最后一个时代的指数。默认: -1.

例子：

```
>>> # Assuming optimizer uses lr = 0.5 for all groups
>>> # lr = 0.05 if epoch < 30
>>> # lr = 0.005 if 30 <= epoch < 80
>>> # lr = 0.0005 if epoch >= 80
>>> scheduler = MultiStepLR(optimizer, milestones=[30,80], gamma=
0.1)
>>> for epoch in range(100):
>>> scheduler.step()
>>> train(...)
>>> validate(...)
```

```
class torch.optim.lr_scheduler.ExponentialLR(optimizer, gamma, last_epoch=-1)
```

将每个参数组的学习速率设置为每一个时代的初始lr衰减。当last\_epoch=-1时，将初始lr设置为lr。

1. optimizer (Optimizer) – 包装的优化器。
2. gamma (float) – 学习率衰减的乘积因子。
3. last\_epoch (int) – 最后一个指数。默认: -1.


```
class torch.optim.lr_scheduler.ReduceLROnPlateau(optimizer, mode='min', factor=0.1, patience=10, verbose=False, threshold=0.0001, threshold_mode='rel', cooldown=0, min_lr=0, eps=1e-08)
```

当指标停止改善时，降低学习率。当学习停滞不前时，模型往往会使学习速度降低2-10倍。这个调度程序读取一个指标量，如果没有提高epochs的数量，学习率就会降低。

1. optimizer (Optimizer) – 包装的优化器。
2. mode (str) – min, max中的一个. 在最小模式下，当监测量停止下降时，lr将减少; 在最大模式下，当监控量停止增加时，会减少。默认值: 'min'。
3. factor (float) – 使学习率降低的因素。new\_lr = lr \* factor. 默认: 0.1.
4. patience (int) – epochs没有改善后，学习率将降低。默认: 10.
5. verbose (bool) – 如果为True，则会向每个更新的stdout打印一条消息。默认: False.
6. threshold (float) – 测量新的最优值的阈值，只关注显著变化。默认: 1e-4.
7. thresholdmode (str) – rel, abs中的一个. 在rel模型, dynamic\_threshold = best ( 1 + threshold ) in 'max' mode or best\_ ( 1 - threshold ) 在最小模型. 在绝对值模型中, dynamic\_threshold = best + threshold 在最大模式或最佳阈值最小模式. 默认: 'rel'.
8. cooldown (int) – 在lr减少后恢复正常运行之前等待的时期数。默认的: 0.
9. min\_lr (float or list) – 标量或标量的列表。对所有的组群或每组的学习速率的一个较低的限制。默认: 0.
10. eps (float) – 适用于lr的最小衰减。如果新旧lr之间的差异小于eps，则更新将被忽略。默认: 1e-8.

```
>>> optimizer = torch.optim.SGD(model.parameters(), lr=0.1, momentum=0.9)
>>> scheduler = torch.optim.ReduceLROnPlateau(optimizer, 'min')
>>> for epoch in range(10):
>>> train(...)
>>> val_loss = validate(...)
>>> # Note that step should be called after validate()
>>> scheduler.step(val_loss)
```

译者署名

用户名	头像	职能	签名
Song		翻译	人生总要追求点什么

## torch.nn.init

```
torch.nn.init.calculate_gain(nonlinearity,param=None)
```

返回给定非线性函数的推荐增益值。值如下：

非线性	获得
linear	1
conv{1,2,3}d	1
sigmoid	1
tanh	5/3
relu	$\sqrt{2}$
leaky_relu	$\sqrt{2/(1+\text{negative\_slope}^2)}$

参数：

1. nonlinearity - 非线性函数（nn.functional名称）
2. param - 非线性函数的可选参数

例子：

```
gain = nn.init.gain('leaky_relu')
```

```
torch.nn.init.uniform(tensor, a=0, b=1)[source]
```

从均匀分布 $U(a, b)$ 中生成值，填充输入的张量或变量

参数：

1. tensor - n维的torch.Tensor或者autograd.Variable
2. a - 均匀分布的下限
3. b - 均匀分布的上限

例子：



```
w = torch.Tensor(3, 5)
print nn.init.uniform(w)
输出:
0.0470 0.9742 0.9736 0.7976 0.1219
0.9390 0.7575 0.9370 0.4786 0.8396
0.1849 0.5384 0.0625 0.3719 0.1739
[torch.FloatTensor of size 3x5]
```

```
torch.nn.init.normal(tensor, mean=0, std=1)
```

从给定均值和标准差的正态分布 $N(\text{mean}, \text{std})$ 中生成值，填充输入的张量或变量

参数：

1. `tensor` –  $n$ 维的`torch.Tensor`或者`autograd.Variable`
2. `mean` – 正态分布的平均值
3. `std` – 正态分布的标准偏差

例子：

```
w = torch.Tensor(3, 5)
print torch.nn.init.normal(w)
```

```
torch.nn.init.constant(tensor, val)
```

使用值`val`填充输入`Tensor`或`Variable`。

参数：

1. `tensor` –  $n$ 维的`torch.Tensor`或`autograd.Variable`
2. `val` – 填充张量的值

例子：

```
w = torch.Tensor(3, 5)
print torch.nn.init.constant(w)
```

```
torch.nn.init.eye(tensor)
```

用单位矩阵来填充2维输入张量或变量。在线性层尽可能多的保存输入特性。

参数：

## 1. tensor – 2维的torch.Tensor或autograd.Variable

例子：

```
w = torch.Tensor(3, 5)
print torch.nn.init.eye(w)
```

```
torch.nn.init.dirac(tensor)
```

用 `Dirac delta` 函数来填充{3, 4, 5}维输入张量或变量。在卷积层尽可能多的保存输入通道特性。

参数：

### 1. tensor – {3, 4, 5}维的torch.Tensor或autograd.Variable

例子：

```
w = torch.Tensor(3, 16, 5, 5)
print torch.nn.init.dirac(w)
```

```
torch.nn.init.xavier_uniform(tensor, gain=1)
```

根据Glorot, X.和Bengio, Y.在"理解难度训练深前馈神经网络"中描述的方法，使用均匀分布，填充张量或变量。结果张量中的值采样自 $U(-a, a)$ ，其中 $a = \text{gain\_sqrt}(2/(\text{fan\_in} + \text{fan\_out}))\_sqrt(3)$ 。该方法也被称为 `glorot` 的初始化。

参数：

1. tensor – n维的torch.Tensor或autograd.Variable
2. gain - 可选的缩放因子

例子：

```
w = torch.Tensor(3, 5)
print torch.nn.init.xavier_uniform(w, gain=nn.init.calculate_gain('relu'))
```

```
torch.nn.init.xavier_normal(tensor, gain=1)
```

根据Glorot, X.和Bengio, Y. 于2010年在"理解难度训练深前馈神经网络"中描述的方法，采用正态分布，填充张量或变量。结果张量中的值采样自均值为0，标准差为 $\text{gain} * \text{sqrt}(2/(\text{fan\_in} + \text{fan\_out}))$ 的正态分布。该方法也被称为 `glorot` 的初始化。

参数：

1. `tensor` –  $n$ 维的`torch.Tensor`或`autograd.Variable`
2. `gain` - 可选的缩放因子

例子：

```
>>> w = torch.Tensor(3, 5)
>>> nn.init.xavier_normal(w)
```

```
torch.nn.init.kaiming_uniform(tensor, a=0, mode='fan_in')
```

根据He, K等人于2015年在"深入研究了超越人类水平的性能：整流器在ImageNet分类"中描述的方法，采用正态分布，填充张量或变量。结果张量中的值采样自 $U(-bound, bound)$ ，其中 $bound = \sqrt{2/((1 + a^2) fan\_in)} \sqrt{3}$ 。该方法也被称为 He 的初始化。

参数：

1. `tensor` –  $n$ 维的`torch.Tensor`或`autograd.Variable`
2. `a` - 此层后使用的整流器的负斜率（默认为ReLU为0）
3. `mode` - "fan\_in"（默认）或"fan\_out"。"fan\_in"保留正向传播时权值方差的量级，"fan\_out"保留反向传播时的量级。

例子：

```
w = torch.Tensor(3, 5)
torch.nn.init.kaiming_uniform(w, mode='fan_in')
```

```
torch.nn.init.kaiming_normal(tensor, a=0, mode='fan_in')
```

根据He, K等人2015年在"深入研究了超越人类水平的性能：整流器在ImageNet分类"中描述的方法，采用正态分布，填充张量或变量。结果张量中的值采样自均值为0，标准差为 $\sqrt{2/((1 + a^2) * fan\_in)}$ 的正态分布。该方法也被称为 He 的初始化。

参数：

1. `tensor` –  $n$ 维的`torch.Tensor`或`autograd.Variable`
2. `a` - 此层后使用的整流器的负斜率（默认为ReLU为0）
3. `mode` - "fan\_in"（默认）或"fan\_out"。"fan\_in"保留正向传播时权值方差的量级，"fan\_out"保留反向传播时的量级。

```
w = torch.Tensor(3, 5)
print torch.nn.init.kaiming_normal(w, mode='fan_out')
```

```
torch.nn.init.orthogonal(tensor, gain=1)
```

使用（半）正交矩阵填充输入张量或变量,参考Saxe, A.等人2013年"深深度线性神经网络学习的非线性动力学的精确解"。输入张量必须至少是2维的,对于更高维度的张量,超出的维度会被展平。

参数：

1. tensor – n维的torch.Tensor或 autograd.Variable，其中 $n \geq 2$
2. gain - 可选缩放因子

例子：

```
w = torch.Tensor(3, 5)
print torch.nn.init.orthogonal(w)
```

```
torch.nn.init.sparse(tensor, sparsity, std=0.01)
```


将二维输入张量或变为稀疏矩阵的非零元素,其中非零元素根据一个均值为0,标准差为std的正态分布生成。如"深度学习通过Hessian免费优化"- Martens, J. (2010)。

参数：

1. tensor – n维的torch.Tensor或autograd.Variable
2. sparsity - 每列中需要被设置成零的元素比例
3. std - 用于生成的正态分布的标准差
4. non-zero values (the) – 例子：

```
w = torch.Tensor(3, 5)
print torch.nn.init.sparse(w, sparsity=0.1)
```

译者署名

用户名	头像	职能	签名
Song		翻译	人生总要追求点什么

# torch.distributions

译者署名

用户名	头像	职能	签名
Song		翻译	人生总要追求点什么

# Multiprocessing 包 - torch.multiprocessing

- 战略管理
- 共享CUDA张量
- 共享策略
  1. 文件描述符 - `file_descriptor`
  2. 文件系统 - `file_system`

`torch.multiprocessing` 是本地 `multiprocessing` 模块的封装。封装了 `multiprocessing` 模块。它注册自定义的 `reducer`，它使用共享内存为不同进程中的相同数据提供视图共享。一旦张量/存储被移动到 `shared_memory`（参见 `sharememory()`），就可以将其发送到其他进程而不进行其它任何操作。

这个API与原始模块100%兼容，将 `import multiprocessing` 改为 `import torch.multiprocessing` 就可以将所有张量通过队列发送或通过其他机制共享转移到共享内存中

由于API的相似性，我们没有记录这个软件包的大部分内容，我们建议您参考 `Python multiprocessing` 原始模块的文档。

警告：如果主进程突然退出（例如因为传入的信号），`Python multiprocessing` 有时无法清理其子进程。这是一个已知的警告，所以如果您在中断解释器后看到任何资源泄漏，这可能意味着这刚刚发生在您身上。

## 战略管理

```
torch.multiprocessing.get_all_sharing_strategies()
```

返回一组当前系统支持的共享策略。

```
torch.multiprocessing.get_sharing_strategy()
```

返回共享CPU张量的当前策略

```
torch.multiprocessing.set_sharing_strategy(new_strategy)
```

设置共享CPU张量的策略

参数: `new_strategy(str)`- 所选策略的名称。应当是上面 `get_all_sharing_strategies()` 中系统支持的共享策略之一。

## 共享CUDA张量

只支持在Python 3中使用 `spawn` 或 `forkserver` 启动方法才支持在进程之间共享CUDA张量。

Python2中的 `multiprocessing` 只能使用 `fork` 创建子进程，并且不支持CUDA运行时。

警告：CUDA API要求导出到其他进程的分配一直保持有效，只要它们被使用。您应该小心，确保您共享的CUDA张量不要超出范围，只要有必要。这不应该是共享模型参数的问题，但传递其他类型的数据应该小心。请注意，此限制不适用于共享CPU内存。

## 共享策略

本节简要概述了不同的共享策略如何工作。请注意，它只适用于CPU张量 - CUDA张量将始终使用CUDA API，因为它们是唯一的共享方式。

### 文件描述符 - `file_descriptor`

注意：这是默认策略（除了不支持的MacOS和OS X）。

此策略将使用文件描述符作为共享内存句柄。当存储器被移动到共享存储器时，每当存储器被移动到共享内存中时，从 `shm_open` 对象获取的文件描述符被缓存，并且当它被发送到其他进程时，文件描述符也将被传送（例如通过UNIX套接字）。接收器还将缓存文件描述符并且 `mmap` 它，以获得对存储数据的共享视图。

请注意，如果要共享很多张量，则此策略将保留大量文件描述符需要很多时间才能打开。如果您的系统对打开的文件描述符数量有限制，并且无法提高，你应该使用 `file_system` 策略。

### 文件系统 - `file_system`

该策略将使用给定的文件名 `shm_open` 来标识共享内存区域。这具有不需要实现缓存从其获得的文件描述符的优点，但是同时容易发生共享内存泄漏。该文件创建后不能被删除，因为其他进程需要访问它以打开其视图。如果进程死机或死机，并且不调用存储析构函数，则文件将保留在系统中。这是非常严重的，因为它们在系统重新启动之前不断使用内存，或者手动释放它们。

为了解决共享内存文件泄漏的问题，`torch.multiprocessing` 将产生一个守护程序 `torch_shm_manager`，它将自己与当前进程组隔离，并且将跟踪所有共享内存分配。一旦连接到它的所有进程退出，它将等待一会儿，以确保不会有新的连接，并且将遍历该组分配的所有共享内存文件。如果发现它们中的任何一个仍然存在，就释放掉它。我们已经测试了这种方法，并且证明对于各种故障是稳健的。不过，如果您的系统具有足够的限制，并且支持 `file_descriptor` 策略，我们不建议切换到该策略。



译者署名

用户名	头像	职能	签名
Song		翻译	人生总要追求点什么

## 分布式通讯包 - torch.distributed

- 基本
- 初始化
- TCP初始化
- 共享文件系统初始化
- 环境变量初始化
- 组
- 点对点通信
- 集体功能

`torch.distributed` 提供了一种类似MPI的接口，用于跨多机器网络交换张量数据。它支持几种不同的后端和初始化方法。

目前，`torch.distributed` 支持三个后端，每个后端具有不同的功能。下表显示哪些功能可用于CPU / CUDA张量。只有当用于构建PyTorch的实现支持它时，MPI才支持cuda。

后端	tcp	gloo	mpi			
设备	中央处理器	GPU	中央处理器	GPU	中央处理器	GPU
---	---	---	---	---	---	---
发送	✓	✗	✗	✗	✓	?
的recv	✓	✗	✗	✗	✓	?
广播	✓	✗	✓	✓	✓	?
all_reduce	✓	✗	✓	✓	✓	?
减少	✓	✗	✗	✗	✓	?
all_gather	✓	✗	✗	✗	✓	?
收集	✓	✗	✗	✗	✓	?
分散	✓	✗	✗	✗	✓	?
屏障	✓	✗	✓	✓	✓	?

### 基本

所述 `torch.distributed` 包提供跨在一个或多个计算机上运行的几个计算节点对多进程并行PyTorch支持与通信原语。该

类`torch.nn.parallel.DistributedDataParallel()`基于此功能，提供同步分布式培训作为围绕任何PyTorch模型的包装器。这不同于所提供的类型的并行的：模

块：`torch.multiprocessing` 和`torch.nn.DataParallel()`在它支持多个网络连接的机器和在用户必须明确地启动主训练脚本的单独副本为每个进程。

在单机同步情况下，`torch.distributed`或

`torch.nn.parallel.DistributedDataParallel()`wrapper可能仍然具有优于数据并行性的其他方法的优点，包括`torch.nn.DataParallel()`：

- 每个进程维护自己的优化器，并且每次迭代执行完整的优化步骤。虽然这可能是冗余的，因为梯度已经被聚集在一起，并且在进程之间进行平均，因此对于每个进程是相同的，这意味着不需要参数广播步骤，从而减少在节点之间传送张量所花费的时间。
- 每个进程都包含一个独立的Python解释器，消除了来自单个Python进程中驱动多个执行线程，模型副本或GPU的额外的解释器开销和“GIL-thrashing”。这对于大量使用Python运行时的模型尤其重要，包括具有循环层或多个小组件的模型。

## 初始化

`torch.distributed.init_process_group()` 在调用任何其他方法之前，需要使用该函数初始化该包。这将阻止所有进程加入。

### `torch.distributed.init_process_group(backend, init_method='env://', kwargs)`

初始化分布式包。

参数：

- `backend` (str) - 要使用的后端的名称。根据构建时配置有效值包括：`tcp`，`mpi` 和 `gloo`。
- `initmethod` (`[str]`(<https://docs.python.org/2/library/functions.html#str>)，\_\_可选\_\_) - 指定如何初始化包的URL。
- `worldsize` (`[int]`(<https://docs.python.org/2/library/functions.html#int>)，\_\_可选\_\_) - 参与作业的进程数。
- `rank` (`int`，\_\_可选\_\_) - 当前进程的等级。
- `groupname` (`[str]`(<https://docs.python.org/2/library/functions.html#str>)，\_\_可选\_\_) - 组名称。请参阅init方法的说明。

### `torch.distributed.get_rank()`

返回当前进程的排名。

Rank是分配给分布式组中每个进程的唯一标识符。它们总是连续的整数，范围从0到 `world_size` 。

## `torch.distributed.get_world_size()`

返回分布式组中的进程数。

---

目前支持三种初始化方式：

### TCP初始化

有两种方法来初始化使用TCP，这两种方法都需要可以从所有进程访问的网络地址和所需的`world_size`。第一种方法需要指定属于等级0进程的地址。第一种初始化方式要求所有进程都具有手动指定的等级。

或者，地址必须是有效的IP多播地址，在这种情况下可以自动分配等级。组播初始化还支持一个`group_name`参数，只要使用不同的组名，就可以为多个作业使用相同的地址。

```
import torch.distributed as dist

Use address of one of the machines
dist.init_process_group(init_method='tcp://10.1.1.20:23456', rank=args.rank, world_size=4)

or a multicast address - rank will be assigned automatically if unspecified
dist.init_process_group(init_method='tcp://[ff15:1e18:5d4c:4cf0:d02d:b659:53ba:b0a7]:23456', world_size=4)
```

### 共享文件系统初始化

另一个初始化方法利用一个文件系统，该文件系统可以从组中的所有计算机共享和可见，以及所需的`world_size`。该URL应`file://`以共享文件系统上的不存在的文件（在现有目录中）开头并包含一个路径。此初始化方法也支持一个`group_name`参数，只要使用不同的组名，就可以使用相同的共享文件路径进行多个作业。

#### 警告

该方法假设文件系统支持使用`fcntl`大多数本地系统进行锁定，NFS支持它。

```
import torch.distributed as dist

Rank will be assigned automatically if unspecified
dist.init_process_group(init_method='file:///mnt/nfs/sharedfile'
, world_size=4,
 group_name=args.group)
```

## 环境变量初始化

该方法将从环境变量中读取配置，从而可以完全自定义获取信息的方式。要设置的变量是：

- MASTER\_PORT - 必需 必须是排名0的机器上的免费端口
- MASTER\_ADDR - 必需（等级0除外）；地址0级节点
- WORLD\_SIZE - 必需 可以在这里设置，也可以调用init函数
- RANK - 必需 可以在这里设置，也可以调用init函数

等级为0的机器将用于设置所有连接。

这是默认方法，这意味着init\_method不必指定（或可以env://）。

## Groups 组

默认集合在默认组（也称为世界）上运行，并要求所有进程进入分布式函数调用。然而，一些工作负载可以从更细粒度的通信中受益。这就是分布式组织发挥作用的地方。[new\\_group\(\)](#)函数可用于创建新组，具有所有进程的任意子集。它返回一个不透明的组句柄，可以作为group参数给予所有集合（集合是分布式函数以在某些众所周知的编程模式中交换信息）。

## torch.distributed.new\_group(\_rank = None)

创建一个新的分布式组。

此功能要求主组中的所有进程（即作为分布式作业的一部分的所有进程）都将输入此功能，即使它们不会是组的成员。此外，应在所有进程中以相同的顺序创建组。

参数：

- ranks (list[int]) - 团体成员名单。返回：分配组的句柄可以被赋予集体调用。

## 点对点通信

## torch.distributed.send(tensor, dst)

同步发送张量。

参数：

- 张量（张量） - 张量发送。
- dst（[int](#)） - 目的地排名。

## torch.distributed.recv(tensor, src=None)

同步接收张量。

参数：

- 张量（[Tensor](#)） - 张量填充收到的数据。
- src（[int](#)） - 源排名。

[isend\(\)](#)并[irecv\(\)](#) 在使用时返回分布式请求对象。一般来说，此对象的类型是未指定的，因为它们不应该手动创建，而是保证支持两种方法：

- [is\\_completed\(\)](#) - 如果操作完成，返回True
- [wait\(\)](#) - 将阻止进程，直到操作完成。 [is\\_completed\(\)](#)保证在返回True后返回True。

## torch.distributed.isend(tensor, dst)

以异步方式发送张量。

参数：

- 张量（张量） - 张量发送。
- dst（[int](#)） - 目的地排名。

返回：

分布式请求对象。

## torch.distributed.irecv(tensor, src)

以异步方式接收张量。

参数：

- 张量（[Tensor](#)） - 张量填充收到的数据。
- src（[int](#)） - 源排名。

返回： 分布式请求对象。

## 集体功能

## **torch.distributed.broadcast(tensor, src, group=<object object="">)</object>**

向整个组广播张量。

`tensor` 在参与集体的所有过程中必须具有相同数量的元素。

参数：

- 张量 ([Tensor](#)) - 如果 `src` 是当前进程的等级，则要发送的数据，否则用于保存接收数据的张量。
- `src` ([int](#)) - 源排名。
- 集团 (可选) - 集体集团。

## **torch.distributed.all\_reduce(tensor, op=<object object="">, group=<object object="">)</object></object>**

减少所有机器上的张量数据，使得所有机器都得到最终结果。

`tensor` 在所有进程中，呼叫将是相同的。

参数：

- 张量 ([张量](#)) - 集体的输入和输出。该功能就地运行。
- `op` (可选) - `torch.distributed.reduce_op` 枚举中的一个值。指定用于元素方式减少的操作。
- 集团 (可选) - 集体集团。

## **torch.distributed.reduce(tensor, dst, op=<object object="">, group=<object object="">)</object></object>**

减少所有机器的张量数据。

只有排名过程 `dst` 才会得到最终结果。

参数：

- 张量 ([张量](#)) - 集体的输入和输出。该功能就地运行。
- `op` (可选) - `torch.distributed.reduce_op` 枚举中的一个值。指定用于元素方式减少的操作。
- 集团 (可选) - 集体集团。

## **torch.distributed.all\_gather(tensor\_list, tensor, group=<object object="">)</object>**

从列表中收集整个组的张量。

参数：

- `tensorlist` (`list` `_[ Tensor ]`) - 输出列表。它应该包含用于集体产出的正确大小的张量。
- 张量 (`Tensor`) - 从当前进程广播的张量。
- 集团 (可选) - 集体集团。

## `torch.distributed.gather(tensor, kwargs)`

在单个过程中收集张量列表。

参数：

- 张量 (`Tensor`) - 输入张量。
- `dst` (`int`) - 目的地排名。除了接收数据之外的所有进程都需要。
- `gatherlist` (`list` `_[ Tensor ]`) - 用于接收数据的适当大小的张量的列表。仅在接收过程中需要。
- 集团 (可选) - 集体集团。

## `torch.distributed.scatter(tensor, kwargs)`

向组中的所有进程散布张量列表。

每个进程将只收到一个张量，并将其数据存储在 `tensor` 参数中。

参数：

- 张量 (`张量`) - 输出张量。
- `src` (`int`) - 源排名。除了发送数据之外的所有进程都需要。
- `scatterlist` (`list` `_[ Tensor ]`) - 要分散的张量列表。只有在发送数据的过程中才需要。
- 集团 (可选) - 集体集团。

## `torch.distributed.barrier(group=<object object=""> </object>)`

同步所有进程。

这个集体阻止进程，直到整个组进入这个功能。

参数：`group` (可选) - Group of the collective.

译者署名



用户名	头像	职能	签名
Song		翻译	人生总要追求点什么

## torch.utils.bottleneck

`torch.utils.bottleneck` 是一个可以用作调试程序瓶颈的初始步骤的工具。它汇总了 Python 分析器和 PyTorch 的 `autograd` 分析器脚本的运行情况。

用命令行运行它

```
python -m torch.utils.bottleneck /path/to/source/script.py [args]
```

[args] 是 `script.py` 中的任意参数，也可以运行如下代码获取更多使用说明。

```
python -m torch.utils.bottleneck -h
```

警告 由于您的脚本将被分析，请确保它在有限的时间内退出。

警告 由于CUDA内核的异步特性，在针对CUDA代码运行时，`cProfile`输出和CPU模式`autograd`分析器可能无法显示以正确的顺序显示：报告的CPU时间报告启动内核所用的时间量，但不包括时间内核花在GPU上执行，除非操作进行同步。在常规CPU模式分析器下，进行同步的Ops似乎非常昂贵。在这些情况下，时序不正确，CUDA模式`autograd`分析器可能会有所帮助。


注意 要决定要查看哪个（纯CPU模式或CUDA模式）`autograd`分析器输出，应首先检查脚本是否受CPU限制（“CPU总时间远远超过CUDA总时间”）。如果它受到CPU限制，查看CPU模式`autograd`分析器的结果将有所帮助。另一方面，如果脚本大部分时间都在GPU上执行，那么开始在CUDA模式`autograd`分析器的输出中查找负责责任的CUDA操作符是有意义的。当然，实际情况要复杂得多，根据你正在评估的模型部分，你的脚本可能不会处于这两个极端之一。如果探查器输出没有帮助，你可以尝试寻找的结果

`torch.autograd.profiler.emit_nvtx()`用`nvprof`。但是，请注意NVTX的开销非常高，并且经常会产生严重偏斜的时间表。

警告 如果您正在分析CUDA代码，那么`bottleneck`运行的第一个分析器（`cProfile`）将在其时间报告中包含CUDA启动时间（CUDA缓冲区分配成本）。如果您的瓶颈导致代码比CUDA启动时间慢得多，这应该没有关系。

有关分析器的更复杂用途（如多 GPU 情况下），请参阅<https://docs.python.org/3/library/profile.html> 或使用 `torch.autograd.profiler.profile()` 获取更多信息。

译者署名

用户名	头像	职能	签名
Song		翻译	人生总要追求点什么

# torch.utils.checkpoint

## torch.utils.checkpoint.checkpoint(function, \*args)

`checkpoint` 模型或模型的一部分

`checkpoint` 通过交换计算内存来工作。而不是存储整个计算图的所有中间激活用于向后计算，`checkpoint` 不会不保存中间激活部分，而是在反向传递中重新计算它们。它可以应用于模型的任何部分。

具体来说，在正向传递中，`function` 将以 `torch.no_grad()` 方式运行，即不存储中间激活。相反，正向传递保存输入元组和 `function` 参数。在向后计算中，保存的输入变量以及 `function` 会被回收，并且正向计算被 `function` 再次计算，现在跟踪中间激活，然后使用这些激活值来计算梯度。

**警告** `checkpoint` 在 `torch.autograd.grad()` 中不起作用，但仅适用于 `torch.autograd.backward()`。

**警告** 如果 `function` 在向后执行和前向执行都不同，例如由于某个全局变量，`checkpoint` 版本将不等同，并且不幸的是无法检测到。

参数：

- `function` - 描述在模型的正向传递或模型的一部分中运行的内容。它也应该知道如何处理作为元组传递的输入。例如，在LSTM中，如果用户通过，应正确使用第一个输入作为第二个输入(activation, hidden)functionactivationhidden
- `args` - 包含输入的元组function

返回：attr function 开 \*args

返回类型：运行输出

## torch.utils.checkpoint.checkpoint\_sequential(functions, segments, \*inputs)

用于 `checkpoint sequential` 模型的辅助函数。

`sequential`模型按顺序执行一系列模块/函数（按顺序）。因此，我们可以将这种模型分为 `checkpoint`。除最后一个段以外的所有段都将以某种 `torch.no_grad()` 方式运行，即不在 `checkpoint` 段的输入，以便在向后传递中重新运行段。

关于 `checkpoint` 如何工作可以参考[checkpoint\(\)](#)。

**警告** `checkpoint` 在 `torch.autograd.grad()` 中不起作用，但仅适用于 `torch.autograd.backward()`。

参数：

- **functions** - A `torch.nn.Sequential`或按顺序运行的模块或函数（包括模型）的列表。
- **segments** - 要在模型中创建的块数
- **segments** - 输入的张量元组**functions**

返回：

`functions` 按顺序运行的输出 `*inputs`

例：

```
>>> model = nn.Sequential(...)
>>> input_var = checkpoint_sequential(model, chunks, input_var)
```

部分地方存在翻译错误，即将修复

译者署名

用户名	头像	职能	签名
Song		翻译	人生总要追求点什么

## torch.utils.cpp\_extension

---

### torch.utils.cpp\_extension.CppExtension(name, sources, args, \*kwargs)

创建一个 C++ 的 `setuptools.Extension` 。

便捷地创建一个 `setuptools.Extension` 具有最小（但通常是足够）的参数来构建 C++ 扩展的方法。

所有参数都被转发给 `setuptools.Extension` 构造函数。

例

```
>>> from setuptools import setup
>>> from torch.utils.cpp_extension import BuildExtension, CppExtension
>>> setup(
 name='extension',
 ext_modules=[
 CppExtension(
 name='extension',
 sources=['extension.cpp'],
 extra_compile_args=['-g']),
],
 cmdclass={
 'build_ext': BuildExtension
 })
```

### torch.utils.cpp\_extension.CUDAExtension(name, sources, args, \*kwargs)

为 CUDA/C++ 创建一个 `setuptools.Extension` 。 创建一个 `setuptools.Extension` 用于构建 CUDA/C ++ 扩展的最少参数（但通常是足够的）的便捷方法。这里包括 CUDA 路径，库路径和运行库。所有参数都被转发给 `setuptools.Extension` 构造函数。

例

```
>>> from setuptools import setup
>>> from torch.utils.cpp_extension import BuildExtension, CppExtension
>>> setup(
 name='cuda_extension',
 ext_modules=[
 CUDAExtension(
 name='cuda_extension',
 sources=['extension.cpp', 'extension_kernel.
cu'],
 extra_compile_args={'cxx': ['-g'],
 'nvcc': ['-O2']})
],
 cmdclass={
 'build_ext': BuildExtension
 })
```

## torch.utils.cpp\_extension.BuildExtension (dist, \*\* kw ) [source]

自定义 setuptools 构建扩展。

setuptools.build\_ext 子类负责传递所需的最小编译器参数（例如 `-std=c++11`）以及混合的 C ++/CUDA 编译（以及一般对 CUDA 文件的支持）。

当使用 BuildExtension 时，它将提供一个用于 `extra_compile_args`（不是普通列表）的词典，通过语言（`cxx` 或 `cuda`）映射到参数列表提供给编译器。这样可以在混合编译期间为 C ++ 和 CUDA 编译器提供不同的参数。

## torch.utils.cpp\_extension.load(name, sources, extra\_cflags=None, extra\_cuda\_cflags=None, extra\_ldflags=None, extra\_include\_paths=None, build\_directory=None, verbose=False)

即时加载(JIT) PyTorch C ++ 扩展。

为了加载扩展，会创建一个 Ninja 构建文件，该文件用于将指定的源编译为动态库。随后将该库作为模块加载到当前 Python 进程中，并从该函数返回，以备使用。

默认情况下，构建文件创建的目录以及编译结果库

是 `<tmp>/torch_extensions/<name>`，其中 `<tmp>` 是当前平台上的临时文件夹以及 `<name>` 为扩展名。这个位置可以通过两种方式被覆盖。首先，如果 `TORCH_EXTENSIONS_DIR` 设置了环境变量，它将替

换 `<tmp>/torch_extensions` 并将所有扩展编译到此目录的子文件夹中。其次，如果 `build_directory` 函数设置了参数，它也将覆盖整个路径，即，库将直接编译到该文件夹中。

要编译源文件，使用默认的系统编译器（C++），可以通过设置 `CXX` 环境变量来覆盖它。将其他参数传递给编译过程，`extra_cflags` 或者 `extra_ldflags` 可以提供。例如，要通过优化来编译您的扩展，你可以传递 `extra_cflags=['-O3']`，也可以使用 `extra_cflags` 传递进一步包含目录。

提供了混合编译的 CUDA 支持。只需将 CUDA 源文件（`.cu` 或 `.cuh`）与其他源一起传递即可。这些文件将被检测，并且使用 `nvcc` 而不是 C++ 编译器进行编译。包括将 `CUDA lib64` 目录作为库目录传递并进行 `cuda` 链接。您可以将其其他参数传递给 `nvcc` `extra_cuda_cflags`，就像使用 C++ 的 `extra_cflags` 一样。使用了各种原始方法来查找 CUDA 安装目录，通常情况下可以正常运行。如果不可以，最好设置 `CUDA_HOME` 环境变量。

- 参数：
  - `name` - 要构建的扩展名。这个必须和 `pybind11` 模块的名字一样！
  - `sources` - C++ 源文件的相对或绝对路径列表。
  - `extra_cflags` - 编译器参数的可选列表，用于转发到构建。
  - `extra_cuda_cflags` - 编译器标记的可选列表，在构建 CUDA 源时转发给 `nvcc`。
  - `extra_ldflags` - 链接器参数的可选列表，用于转发到构建。
  - `extra_include_paths` - 转发到构建的包含目录的可选列表。
  - `build_directory` - 可选路径作为构建区域。
  - `verbose` - 如果为 `True`，打开加载步骤的详细记录。
- 返回：
  - 加载 PyTorch 扩展作为 Python 模块。

例

```
>>> from torch.utils.cpp_extension import load
>>> module = load(
 name='extension',
 sources=['extension.cpp', 'extension_kernel.cu'],
 extra_cflags=['-O2'],
 verbose=True)
```

## torch.utils.cpp\_extension.include\_paths(cuda=False)

获取构建 C++ 或 CUDA 扩展所需的库路径。

- 参数： `cuda` - 如果为 `True`，则包含 CUDA 特定的包含路径。
- 返回： 包含路径字符串的列表。

例如：



```
from setuptools import setup
from torch.utils.cpp_extension import BuildExtension, CppExtension

torch.utils.cpp_extension.include_paths(cuda=False)
['/usr/local/lib/python3.6/site-packages/torch/lib/include', '
/usr/local/lib/python3.6/site-packages/torch/lib/include/TH', '/
usr/local/lib/python3.6/site-packages/torch/lib/include/THC']
```

## torch.utils.cpp\_extension.check\_compiler\_abi\_compatibility(compiler)

验证给定的编译器是否与 PyTorch ABI兼容。

- 参数：compiler(str) - 要检查可执行的编译器文件名(例如g++),必须在 shell 进程中可执行。
- 返回：如果编译器（可能）与 PyTorch ABI不兼容，则为 False，否则返回 True。

## torch.utils.cpp\_extension.verify\_ninja\_availability()

如果可以在ninja上运行则返回 True。

文档地址：[\[torch.utils.cpp\\_extension](#)

]([https://ptorch.com/docs/8/torch-utils-cpp\\_extension](https://ptorch.com/docs/8/torch-utils-cpp_extension))

### 译者署名

用户名	头像	职能	签名
Song		翻译	人生总要追求点什么

## torch.utils.data

```
class torch.utils.data.Dataset
```

表示数据集的抽象类。

所有其他数据集都应该进行子类化。所有子类应该覆盖 `__len__` 和 `__getitem__`，`__len__` 提供了数据集的大小，`__getitem__` 支持整数索引，范围从0到`len(self)`。

```
class torch.utils.data.TensorDataset(data_tensor, target_tensor)
```

包装数据和目标张量的数据集。

通过沿着第一个维度索引两个张量来恢复每个样本。

参数：

1. `data_tensor` (Tensor) — 包含样本数据
2. `target_tensor` (Tensor) — 包含样本目标（标签）

例子：

```
x = torch.linspace(1, 10, 10) # x data (torch tensor)
y = torch.linspace(10, 1, 10) # y data (torch tensor)

先转换成 torch 能识别的 Dataset
torch_dataset = torch.utils.data.TensorDataset(data_tensor=x, target_tensor=y)
```

```
class torch.utils.data.ConcatDataset(datasets)
```

连接多个数据集。

目的：组合不同的现有数据集，可能是大规模数据集，因为连续操作是随意连接的。

- `datasets` 的参数：要连接的数据集列表
- `datasets` 样式：iterable

```
class torch.utils.data.DataLoader(dataset, batch_size=1, shuffle=False, sampler=None, num_workers=0, collate_fn=<function default_collate>, pin_memory=False, drop_last=False)
```

数据加载器。组合数据集和采样器，并在数据集上提供单进程或多进程迭代器。

参数：

1. `dataset (Dataset)` – 从中加载数据的数据集。
2. `batch_size (int, optional)` – 批训练的数据个数(默认: 1)。
3. `shuffle (bool, optional)` – 设置为True在每个epoch重新排列数据（默认值: False, 一般打乱比较好）。
4. `sampler (Sampler, optional)` – 定义从数据集中提取样本的策略。如果指定，则忽略`shuffle`参数。
5. `batch_sampler (sampler, 可选)` - 和`sampler`一样，但一次返回一批索引。与`batch_size`，`shuffle`，`sampler`和`drop_last`相互排斥。
6. `num_workers (int, 可选)` – 用于数据加载的子进程数。0表示数据将在主进程中加载（默认值: 0）
7. `collate_fn (callable, optional)` – 合并样本列表以形成小批量。
8. `pin_memory (bool, optional)` – 如果为True，数据加载器在返回前将张量复制到CUDA固定内存中。
9. `drop_last (bool, optional)` – 如果数据集大小不能被`batch_size`整除，设置为True可删除最后一个不完整的批处理。如果设为False并且数据集的大小不能被`batch_size`整除，则最后一个batch将更小。(默认: False)

```
class torch.utils.data.sampler.Sampler(data_source)
```

所有采样器的基础类。

每个采样器子类必须提供一个 `__iter__` 方法，提供一种迭代数据集元素的索引的方法，以及返回迭代器长度的 `__len__` 方法。

```
class torch.utils.data.sampler.SequentialSampler(data_source)
```

始终以相同的顺序将样本元素按顺序排列。

参数：

- `data_source (Dataset)` – 采样的数据集。

```
class torch.utils.data.sampler.RandomSampler(data_source)
```

样本元素随机排列，并没有替换。

参数：- `data_source (Dataset)` – 采样的数据集。

```
class torch.utils.data.sampler.SubsetRandomSampler(indices)
```

样本元素从指定的索引列表中随机抽取，并没有替换。

参数： - `indices (list)` - 索引的列表

```
class torch.utils.data.sampler.WeightedRandomSampler(weights, num_samples, replacement=True)
```

样本元素来自 `[0,...,len(weights)-1]`，给定概率(权重)。

参数：

- `weights (list)` - 权重列表。不需要加起来为1
- `num_samples (int)` - 要绘制的样本数

```
class torch.utils.data.distributed.DistributedSampler(dataset, num_replicas=None, rank=None)
```

将数据加载限制到数据集的子集的采样器。


在 `torch.nn.parallel.DistributedDataParallel` 中是特别有用的。在这种情况下，每个进程都可以作为 `DataLoader` 采样器传递一个 `DistributedSampler` 实例，并加载独占的原始数据集的子集。

注意 假设数据集的大小不变。

参数：

- `dataset` - 采样的数据集。
- `num_replicas(可选)` - 参与分布式培训的进程数。
- `rank(可选)` - `num_replicas` 中当前进程的等级。

## 译者署名

用户名	头像	职能	签名
Song		翻译	人生总要追求点什么

# torch.utils.ffi

```
torch.utils.ffi.create_extension(name, headers, sources, verbose=True, with_cuda=False, package=False, relative_to='.', **kwargs)
```

创建并配置一个构建PyTorch扩展的cffi.FFI对象。

参数：

- 1. **name (str)** – 包名。可以是嵌套的模块如 `.ext.my_lib`。
- 2. **headers (str or List[str])** – 仅包含导出函数的头文件列表
- 3. **sources (List[str])** – 要编译的源列表。
- 4. **verbose (bool, optional)** – 如果设置为 `False`，则不打印任何输出（默认值：`True`）。
- 5. **with\_cuda (bool, optional)** – 设置为 `True` 表示使用CUDA头进行编译（默认值：`False`）。
- 6. **package (bool, optional)** – 设置为 `True` 表示以程序包的模式构建（对于要以pip包安装的模块）（默认值：`False`）。
- 7. **relative\_to (str, optional)** – 构建文件的路径。 `package` 为 `True` 时需要。最好使用 `__file__` 作为参数。
- 8. **kwargs** – 传递给ffi以声明扩展名的其他参数。详情请参阅[扩展API参考](#)。

## 译者署名

用户名	头像	职能	签名
Song		翻译	人生总要追求点什么

# torch.utils.model\_zoo

```
torch.utils.model_zoo.load_url(url, model_dir=None)
```

在给定URL上加载Torch序列化对象。

如果对象已经存在于 `model_dir` 中，则将被反序列化并返回。URL的文件名部分应遵循命名约定 `filename-<sha256>.ext`，其中 `<sha256>` 是文件内容的哈希(SHA256)的前八位或更多位数字。哈希用于确保名称唯一性的并验证文件的内容。

`model_dir` 的默认值为 `$TORCH_HOME/models`，其中 `$TORCH_HOME` 默认为 `~/.torch`。可以使用 `$TORCH_MODEL_ZOO` 环境变量来覆盖默认目录。

参数：

- `url (string)` - 要下载对象的URL
- `model_dir (string, 可选)` - 保存对象的目录
- `map_location(可选)` - 指定如何重映射存储位置的函数或dict（参见 `torch.load`）

例如：

```
>>> state_dict = torch.utils.model_zoo.load_url('https://s3.amazonaws.com/pytorch/models/resnet18-5c106cde.pth')
```

## 译者署名

用户名	头像	职能	签名
Song		翻译	人生总要追求点什么

## torch.onnx

`torch.onnx` 模块包含将模型导出为 ONNX IR 格式的功能。这些模型可以加载 ONNX 库，然后转换为在其他深度学习框架上运行的模型。



ONNX  
ptorch.com

### 示例：从PyTorch到Caffe2的端到端的AlexNet

这是一个简单的脚本，将 `torchvision` 中定义的预训练的 AlexNet 导出到 ONNX 中。它运行一轮推理，然后将结果跟踪模型保存到 `alexnet.proto`：

```
from torch.autograd import Variable
import torch.onnx
import torchvision

dummy_input = Variable(torch.randn(1, 3, 224, 224)).cuda()
model = torchvision.models.alexnet(pretrained=True).cuda()
torch.onnx.export(model, dummy_input, "alexnet.proto", verbose=True)
```

保存文件 `alexnet.proto` 是一个二进制 `protobuf` 文件，其中包含您导出的模型（在本例中为 AlexNet）的网络结构和参数。关键字参数 `verbose=True` 导致导出器打印出一个人类可读的网络表示：

```

All parameters are encoded explicitly as inputs. By convention,
learned parameters (ala nn.Module.state_dict) are first, and the
actual inputs are last.
graph(%1 : Float(64, 3, 11, 11)
 %2 : Float(64)
 # The definition sites of all variables are annotated with
 type
 # information, specifying the type and size of tensors.
 # For example, %3 is a 192 x 64 x 5 x 5 tensor of floats.
 %3 : Float(192, 64, 5, 5)
 %4 : Float(192)
 # ---- omitted for brevity ----
 %15 : Float(1000, 4096)
 %16 : Float(1000)
 %17 : Float(10, 3, 224, 224)) { # the actual input!
 # Every statement consists of some output tensors (and their types),
 # the operator to be run (with its attributes, e.g., kernels,
 # strides,
 # etc.), its input tensors (%17, %1)
 %19 : UNKNOWN_TYPE = Conv[kernels=[11, 11], strides=[4, 4], pads=[2, 2, 2, 2], dilations=[1, 1], group=1](%17, %1), uses = [%20.i0]];
 # UNKNOWN_TYPE: sometimes type information is not known. We hope to eliminate
 # all such cases in a later release.
 %20 : Float(10, 64, 55, 55) = Add[broadcast=1, axis=1](%19, %21.i0);
 %21 : Float(10, 64, 55, 55) = Relu(%20), uses = [%22.i0];
 %22 : Float(10, 64, 27, 27) = MaxPool[kernels=[3, 3], pads=[0, 0, 0, 0], dilations=[1, 1], strides=[2, 2]](%21), uses = [%23.i0];
 # ...
 # Finally, a network returns some tensors
 return (%58);
}

```

您也可以使用 [onnx](#) 库来验证 `protobuf` 。您可以 `onnx` 用 `conda` 安装：

```
conda install -c conda-forge onnx
```

然后，您可以运行：



```
import onnx

Load the ONNX model
model = onnx.load("alexnet.proto")

Check that the IR is well formed
onnx.checker.check_model(model)

Print a human readable representation of the graph
onnx.helper.printable_graph(model.graph)
```

要用 `caffe2` 运行导出的脚本，您将需要三件事情：

- 1、您需要安装 `Caffe2` 。如果您还没有，请按照安装说明进行操作。
- 2、你需要 `onnx-caffe2` ，一个纯 `Python` 库，为 `ONNX` 提供一个 `Caffe2` 后端。 `onnx-caffe2` 你可以用 `pip` 来安装：

```
pip install onnx-caffe2
```

安装完成后，您可以使用 `Caffe2` 的后端：

```
...continuing from above
import onnx_caffe2.backend as backend
import numpy as np

rep = backend.prepare(model, device="CUDA:0") # or "CPU"
For the Caffe2 backend:
rep.predict_net is the Caffe2 protobuf for the network
rep.workspace is the Caffe2 workspace for the network
(see the class onnx_caffe2.backend.Workspace)
outputs = rep.run(np.random.randn(10, 3, 224, 224).astype(np.float32))
To run networks with more than one input, pass a tuple
rather than a single numpy ndarray.
print(outputs[0])
```

未来，其他框架也会有后端。

## 限制与不足

- `ONNX` 出口是一个基于跟踪的出口，这意味着它通过执行一次模型来运行，并导出在此运行期间实际运行的运营商。这意味着如果您的模型是动态的，例如，根据输入数据改变行为，则导出将不准确。类似地，跟踪可能仅对特定输入大小有效（这是我们为什么需要明确输入跟踪的一个原因）。我们建议检查模型跟踪并确保跟踪的运算符看起来合理。

- `PyTorch` 和 `Caffe2` 通常具有一些数字差异的操作符的实现。根据模型结构的不同，这些差异可能是微不足道的，但是它们也会造成行为上的重大分歧（特别是在未经训练的模型上）。在未来的版本中，我们计划让 `Caffe2` 直接调用操作员的 `Torch` 实现，以帮助您在精度很重要的时候处理这些差异，并记录这些差异。

## 支持的运算函数

支持以下运算符：

```
add (nonzero alpha not supported), sub (nonzero alpha not supported),
mul, div, cat, mm, addmm, neg, tanh, sigmoid, mean, t, expand,
transpose, view, split, squeeze, prelu (single weight shared amon
, threshold (non-zero threshold/non-zero value not supported), leaky_
, glu, softmax, avg_pool2d (ceil_mode not supported), log_softmax,
, elu, Conv, BatchNorm, MaxPool1d (ceil_mode not supported), MaxPoo
, MaxPool3d (ceil_mode not supported), Embedding (no optional argumen
, RNN, ConstantPadNd, Dropout, FeatureDropout (training mode not sup
, Index (constant integer and tuple indices supported), Negate
```

以上设置的操作足以导出以下型号：

```
AlexNet, DCGAN, DenseNet, Inception (warning: this model is highly
, ResNet, SuperResolution, VGG, word_language_model
```

用于指定运算符定义的界面是高度实验性的，并且没有记录；有冒险精神的用户应该注意，这些 `API` 可能会在未来的界面中发生变化。

## torch.onnx功能

```
torch.onnx.export(model, args, f, export_params=True, verbose=False,
training=False)
```

将模型导出为 `ONNX` 格式。这个导出器运行你的模型一次，以获得其导出的执行轨迹；目前，它不支持动态模型（例如，`RNN`）。

另见：`onnx-export`

### 参数：

- 模型（`torch.nn.Module`） - 要导出的模型。
- `args`（参数元组） - 模型的输入，例如，这 `-model(*args)` 是模型的有效调用。任何非变量参数将被硬编码到导出的模型中；任何变量参数都将成为输出模型的输入，按照它们在参数中出现的顺序。如果 `args` 是一个变量，这相当于用该变量的一个元组来调用它。（注意：将关键字参数传递给模型目前还不支持，如果需要，给我们留言。）

- **f** - 类文件对象（必须实现返回文件描述符的`fileno`）或包含文件名的字符串。一个二进制Protobuf将被写入这个文件。
- **export\_params**（布尔，默认为True） - 如果指定，所有参数将被导出。如果要导出未经训练的模型，请将其设置为False。在这种情况下，导出的模型将首先将其所有参数作为参数，按照指定的顺序`model.state_dict().values()`
- **verbose**（布尔，默认为False） - 如果指定，我们将打印出一个调试描述的导出轨迹。
- **training**（布尔，默认为False） - 在训练模式下导出模型。目前，ONNX只是为了推导出模型，所以你通常不需要将其设置为True。

## 译者署名

用户名	头像	职能	签名
Song		翻译	人生总要追求点什么

# 遗留包 - torch.legacy

---

此包中包含从Lua Torch移植来的代码。

为了可以使用现有的模型并且方便当前Lua Torch使用者过渡，我们创建了这个包。可以在 torch.legacy.nn 中找到 nn 代码，并在 torch.legacy.optim 中找到 optim 代码。API应该完全匹配Lua Torch。

## 译者署名

用户名	头像	职能	签名
Song		翻译	人生总要追求点什么

## torchvision 参考

---

# torchvision

`torchvision` 包朗阔了目前流行的数据集，模型结构和常用的图片转换工具。  
如下代码用于获取加载图像的包的名称。

```
torchvision.get_image_backend()
```

指定用于加载图像的包。

```
torchvision.set_image_backend(backend)
```

参数：`backend (string)` -`backend`代表图片名称。属于{'pil ','accimage'}之一。  
`accimage`包使用Intel IPP库。速度比 PIL快，但不支持多操作。

## 译者署名

用户名	头像	职能	签名
Song		翻译	人生总要追求点什么

# torchvision.datasets

`torchvision.datasets` 中包含了以下数据集

1. [MNIST](#)
2. [COCO](#)（用于图像标注和目标检测）(Captioning and Detection)
3. [LSUN Classification](#)
4. [ImageFolder](#)
5. [Imagenet-12](#)
6. [CIFAR10 and CIFAR100](#)
7. [STL10](#)
8. [SVHN](#)
9. [PhotoTour](#)

所有数据集都是 `torch.utils.data.Dataset` 的子类，即它们具有 `getitem` 和 `len` 实现方法。因此，它们都可以传递给 `torch.utils.data.DataLoader` 可以使用 `torch.multiprocessing` 工作人员并行加载多个样本的数据。例如：

```
imagenet_data = torchvision.datasets.ImageFolder('path/to/imagenet_root/')
data_loader = torch.utils.data.DataLoader(imagenet_data,
 batch_size=4,
 shuffle=True,
 num_workers=args.nThreads)
```

所有的数据集都有几乎相似的API。他们都有两个共同的参数：`transform`和`target_transform`分别转换输入和目标。

## MNIST

```
dset.MNIST(root, train=True, transform=None, target_transform=None, download=False)
```

参数说明:

- `root` : 数据集，存在于根目录 `processed/training.pt` 和 `processed/test.pt` 中。
- `train` : `True` = 训练集, `False` = 测试集
- `download` : 如果为 `true`，请从Internet下载数据集并将其放在根目录中。如果数据集已经下载，则不会再次下载。

- `transform` : 接收PIL映像并返回转换版本的函数/变换。例如: `transforms.RandomCrop`
- `target_transform` : 一个接收目标并转换它的函数/变换。

## COCO

需要安装COCO API

```
dset.CocoCaptions(root="dir where images are", annFile="json annotation file", [transform, target_transform])
```

参数说明：

- `root (string)` - 图像下载到的根目录。
- `annFile (string)` - json注释文件的路径。
- `transform (可调用, 可选)` - 接收PIL映像并返回转换版本的函数/变换。例如: `transforms.ToTensor`
- `target_transform (可调用, 可选)` - 一个接收目标并转换它的函数/变换。

例子:

```
import torchvision.datasets as dset
import torchvision.transforms as transforms
cap = dset.CocoCaptions(root = 'dir where images are',
 annFile = 'json annotation file',
 transform=transforms.ToTensor())

print('Number of samples: ', len(cap))
img, target = cap[3] # load 4th sample

print("Image Size: ", img.size())
print(target)
```

输出:

```
Number of samples: 82783
Image Size: (3L, 427L, 640L)
[u'A plane emitting smoke stream flying over a mountain.',
u'A plane darts across a bright blue sky behind a mountain covered in snow',
u'A plane leaves a contrail above the snowy mountain top.',
u'A mountain that has a plane flying overhead in the distance.',
,
u'A mountain view with a plume of smoke in the background']
```



**getitem(index)** 参数：index(int) - 索引 返回：元组（图像，目标）。目标是图片的标题列表。 返回类型：元组

检测：

```
dset.CocoDetection(root="dir where images are", annFile="json an
notation file", [transform, target_transform])
```

参数：

- root (string) - 图像下载到的根目录。
- annFile (string) - json注释文件的路径。
- transform (可调用，可选) - 接收PIL映像并返回转换版本的函数/变换。例如，transforms.ToTensor
- target\_transform (可调用，可选) - 一个接收目标并转换它的函数/变换。

**getitem(index)** 参数：index(int) - 索引 返回：元组（图像，目标）。目标是图片的标题列表。 返回类型：元组

## LSUN

```
dset.LSUN(db_path, classes='train', [transform, target_transform
])
```

参数说明：

- db\_path = 数据集文件的根目录
- classe = train (所有类别, 训练集), val (所有类别, 验证集), test (所有类别, 测试集) [ bedroom\_train , church\_train ,...]: 要加载的类别列表
- transform (可调用，可选) - 接收PIL映像并返回转换版本的函数/变换。例如，transforms.RandomCrop
- target\_transform (可调用，可选) - 一个接收目标并转换它的函数/变换。

## ImageFolder

一个通用的数据加载器，数据集中的数据以以下方式组织

```

root/dog/xxx.png
root/dog/xyx.png
root/dog/xxz.png

root/cat/123.png
root/cat/nsdf3.png
root/cat/asd932_.png
dset.ImageFolder(root="root folder path", [transform, target_transform])

```

参数说明:

- `root` (string) - 根目录路径。
- `transform` (可调用, 可选) - 接收PIL映像并返回转换版本的函数/变换。例如, `transforms.RandomCrop`
- `target_transform` (可调用, 可选) - 一个接收目标并转换它的函数/变换。
- `loader` - 加载给定其路径的图像的函数。

## Imagenet-12

这应该简单地用 `ImageFolder` 数据集实现。数据按照这里所述进行预处理 [这里有一个例子](#)

## CIFAR

```

dset.CIFAR10(root, train=True, transform=None, target_transform=None, download=False)

dset.CIFAR100(root, train=True, transform=None, target_transform=None, download=False)

```

参数说明：

- `root` : cifar-10-batches-py 的根目录
- `train` : `True` = 训练集, `False` = 测试集
- `download` : `True` = 从互联网上下载数据, 并将其放在`root`目录下。如果数据集已经下载, 什么都不干。
- `transform` (可调用, 可选) - 接收PIL映像并返回转换版本的函数/变换。例如, `transforms.RandomCrop`
- `target_transform` (可调用, 可选) - 一个接收目标并转换它的函数/变换。

## STL10

```
dset.STL10(root, split='train', transform=None, target_transform=
None, download=False)
```

参数说明：

- **root** : stl10\_binary的根目录
- **split** : 'train' = 训练集, 'test' = 测试集, 'unlabeled' = 无标签数据集, 'train+unlabeled' = 训练 + 无标签数据集 (没有标签的标记为-1)
- **download** : True = 从互联网上下载数据，并将其放在root目录下。如果数据集已经下载，什么都不干。
- **transform** (可调用，可选) - 接收PIL映像并返回转换版本的函数/变换。例如，transforms.RandomCrop
- **target\_transform** (可调用，可选) - 一个接收目标并转换它的函数/变换。

## SVHN

```
class torchvision.datasets.SVHN(root, split='train', transform=None,
target_transform=None, download=False)
```

参数：

- **root** (string) - 目录SVHN存在的数据集的根目录。
- **split** (string) - {'train', 'test', 'extra'}之一。因此选择数据集。“额外”是额外的训练集。
- **transform** (可调用，可选) - 接收PIL映像并返回转换版本的函数/变换。例如，transforms.RandomCrop
- **target\_transform** (可调用，可选) - 一个接收目标并转换它的函数/变换。
- **download** (bool，可选) - 如果为true，请从Internet下载数据集并将其放在根目录中。如果数据集已经下载，则不会再次下载。


## PhotoTour

```
class torchvision.datasets.PhotoTour(root, name, train=True, transform=None,
download=False)
```

参数：

- **root** (string) - 图像所在的根目录。
- **name** (string) - 要加载的数据集的名称。
- **transform** (可调用，可选) - 接收PIL映像并返回转换版本的函数/变换。
- **download** (bool，可选) - 如果为true，请从Internet下载数据集并将其放在根目录中。如果数据集已经下载，则不会再次下载。

译者署名

用户名	头像	职能	签名
Song		翻译	人生总要追求点什么

## torchvision.models

`torchvision.models` 模块的子模块中包含以下模型结构。

- [AlexNet](#)
- [VGG](#)
- [ResNet](#)
- [SqueezeNet](#)
- [DenseNet](#)

可以通过调用构造函数来构造具有随机权重的模型：

```
import torchvision.models as models
resnet18 = models.resnet18()
alexnet = models.alexnet()
squeezenet = models.squeezenet1_0()
densenet = models.densenet_161()
```

我们提供的Pathway变体和alexnet预训练的模型，利用pytorch的 `torch.utils.model_zoo` 。这些可以通过构建 `pretrained=True` ：

```
import torchvision.models as models
resnet18 = models.resnet18(pretrained=True)
alexnet = models.alexnet(pretrained=True)
```

所有预训练的模型的期望输入图像相同的归一化，即小批量形状通道的RGB图像（ $3 \times H \times W$ ），其中H和W预计将至少224。这些图像必须被加载到 $[0, 1]$ 的范围内，然后使用平均= $[0.485, 0.456, 0.406]$ 和STD= $[0.229, 0.224, 0.225]$ 进行归一化。您可以使用以下转换来正常化：

```
normalize = transforms.Normalize(mean=[0.485, 0.456, 0.406], std=[0.229, 0.224, 0.225])
```

这种规范化的一个例子可以在这里找到ImageNet的例子 ImageNet 1-crop错误率（224x224）

Network	Top-1 error	Top-5 error
ResNet-18	30.24	10.92
ResNet-34	26.70	8.58
ResNet-50	23.85	7.13
ResNet-101	22.63	6.44
ResNet-152	21.69	5.94
Inception v3	22.55	6.44
AlexNet	43.45	20.91
VGG-11	30.98	11.37
VGG-13	30.07	10.75
VGG-16	28.41	9.62
VGG-19	27.62	9.12
SqueezeNet 1.0	41.90	19.58
SqueezeNet 1.1	41.81	19.38
Densenet-121	25.35	7.83
Densenet-169	24.00	7.00
Densenet-201	22.80	6.43
Densenet-161	22.35	6.20

```
torchvision.models.alexnet(pretrained=False, ** kwargs)
```

AlexNet 模型结构 [paper地址](#)

`pretrained (bool)` – True, 返回在 ImageNet 上训练好的模型。

```
torchvision.models.resnet18(pretrained=False, ** kwargs)
```

构建一个 resnet18 模型 `pretrained (bool)` – True, 返回在 ImageNet 上训练好的模型。

```
torchvision.models.resnet34(pretrained=False, ** kwargs)
```

构建一个 ResNet-34 模型. Parameters: `pretrained (bool)` – True, 返回在 ImageNet 上训练好的模型。

```
torchvision.models.resnet50(pretrained=False, ** kwargs)
```

构建一个ResNet-50模型

pretrained (bool) – True, 返回在ImageNet上训练好的模型。

```
torchvision.models.resnet101(pretrained=False, ** kwargs)
```

构建一个ResNet-101 模型。

pretrained (bool) – True, 返回在ImageNet上训练好的模型。

```
torchvision.models.resnet152(pretrained=False, ** kwargs)
```

构建一个ResNet-152 模型。

pretrained (bool) – True, 返回在ImageNet上训练好的模型。

```
torchvision.models.vgg11(pretrained=False, ** kwargs)
```

VGG 11-layer model (configuration “A”) -

pretrained (bool) – True, 返回在ImageNet上训练好的模型。

```
torchvision.models.vgg11_bn(** kwargs)
```

VGG 11-layer model (configuration “A”) with batch normalization

```
torchvision.models.vgg13(pretrained=False, ** kwargs)
```

VGG 13-layer model (configuration “B”)

pretrained (bool) – True, 返回在ImageNet上训练好的模型。

```
torchvision.models.vgg13_bn(** kwargs)
```

VGG 13-layer model (configuration “B”) with batch normalization

```
torchvision.models.vgg16(pretrained=False, ** kwargs)
```

### VGG 16-layer model (configuration “D”)

Parameters: pretrained (bool) – If True, returns a model pre-trained on ImageNet

```
torchvision.models.vgg16_bn(**kwargs)
```

### VGG 16-layer model (configuration “D”) with batch normalization

```
torchvision.models.vgg19(pretrained=False, **kwargs)
```

### VGG 19-layer model (configuration “E”)

pretrained (bool) – True, 返回在ImageNet上训练好的模型。

```
torchvision.models.vgg19_bn(**kwargs)
```

### VGG 19-layer model (configuration ‘E’) with batch normalization

#### 译者署名

用户名	头像	职能	签名
Song		翻译	人生总要追求点什么



## torchvision.transform

1. 对PIL.Image进行变换
2. 对Tensor进行变换
3. Conversion Transforms
4. 通用变换

变换是常用的图像变换。它们可以用 `Compose` 连接在一起。

```
class torchvision.transforms.Compose(transforms)
```

将多个transform组合起来使用。

`transforms`：由transform构成的列表. 例子：

```
transforms.Compose([
 transforms.CenterCrop(10),
 transforms.ToTensor(),
])
```

### 对PIL.Image进行变换

```
class torchvision.transforms.Scale(size, interpolation=2)
```

按照规定的尺寸重新调节 `PIL.Image`。

参数说明：

1. `size (sequence or int)` - 期望输出尺寸。如果`size`是一个像(w, h)的序列，输出大小将按照w,h匹配到。如果大小是int，则图像将匹配到这个数字。例如，如果原图的 `height>width` ,那么改变大小后的图片大小是 `(size*height/width, size)`。
2. `interpolation (int, optional)` -需要添加值。默认的是 `PIL.Image.BILINEAR`

```
class torchvision.transforms.CenterCrop(size)
```

将给定的PIL.Image进行中心切割，得到给定的size，size可以是tuple，(target\_height, target\_width)。size也可以是一个Integer，在这种情况下，切出来的图片的形状是正方形。

```
class torchvision.transforms.RandomCrop(size, padding=0)
```

切割中心点的位置随机选取。size可以是tuple也可以是Integer。

```
class torchvision.transforms.RandomHorizontalFlip
```

随机水平翻转给定的PIL.Image,概率为0.5。即：一半的概率翻转，一半的概率不翻转。

```
class torchvision.transforms.RandomSizedCrop(size, interpolation=2)
```

先将给定的PIL.Image随机切，然后再resize成给定的size大小。

```
class torchvision.transforms.Pad(padding, fill=0)
```

将给定的PIL.Image的所有边用给定的pad value填充。padding：要填充多少像素  
fill：用什么值填充

## 对Tensor进行变换

---

```
class torchvision.transforms.Normalize(mean, std)
```

给定均值：(R,G,B) 方差：(R, G, B)，将会把Tensor正则化。即：  
 $\text{Normalized\_image} = (\text{image} - \text{mean}) / \text{std}$ 。

参数说明：

1. mean (sequence) – 序列R, G, B的均值。
2. std (sequence) – 序列 R, G, B 的平均标准偏差。

**\*\*call(tensor)\*\*** 参数: tensor (Tensor) – 规范化的大小 (c, h, w) 的张量图像. 返回结果: 规范化的图片. 返回样式: Tensor张量

## 对Conversion进行变换

---

```
class torchvision.transforms.ToTensor
```

把一个取值范围是[0,255]的PIL.Image或者shape为(H,W,C)的numpy.ndarray，转换成形状为[C,H,W]，取值范围是[0,1.0]的torch.FloatTensor

**\*\*call(pic)\*\***

1. 参数: pic (PIL.Image or numpy.ndarray) – 图片转换为张量.
2. 返回结果: 转换后的图像。
3. 返回样式: Tensor张量

```
class torchvision.transforms.ToPILImage
```

将shape为(C,H,W)的Tensor或shape为(H,W,C)的numpy.ndarray转换成PIL.Image，值不变。

**\*\*call(pic)\*\***

1. 参数: pic (Tensor or numpy.ndarray) – 图像转换为pil.image。
2. 返回结果: 图像转换为PIL.Image.
3. 返回样式: PIL.Image

## 通用变换

```
class torchvision.transforms.Lambda(lambd)
```

使用lambd作为转换器。

参数说明: lambd (function) – Lambda/function 用于转换.

## 译者署名

用户名	头像	职能	签名
Song		翻译	人生总要追求点什么



## torchvision.utils

```
torchvision.utils.make_grid(tensor, nrow=8, padding=2, normalize=False, range=None, scale_each=False, pad_value=0)
```

制作图像网格。

参数说明:

1. **tensor (Tensor or list)** – 4D小批量形状Tensor张量 ( $B \times C \times H \times W$ ) 或所有大小相同的图像列表。
2. **nrows (int, optional)** – 网格中的行数。最终的网格大小 ( $B / \text{nrow}$ ,  $\text{nrow}$ )。默认值是8。
3. **normalize (bool, optional)** – 如果TRUE, 将图像移到范围  $(0, 1)$  中, 减去最小值并除以最大像素值。
4. **range (tuple, optional)** – 元组 ( $\text{min}$ ,  $\text{max}$ ), 其中 $\text{min}$ 和 $\text{max}$ 是数字, 然后使用这些数字对图像进行标准化。默认情况下,  $\text{min}$ 和 $\text{max}$ 是从张量计算的。
5. **scale\_each (bool, optional)** – 如果TRUE, 则在批处理图像中分别缩放每个图像, 而不是在所有图像上 (最小、最大) 缩放图像。
6. **pad\_value (float, optional)** – 填充像素的值。


查看下面的例子：

```
torchvision.utils.save_image(tensor, filename, nrow=8, padding=2, normalize=False, range=None, scale_each=False, pad_value=0)
```

将给定的Tensor张量保存到图像文件中。参数说明:

1. **Tensor (张量或者列表)** – 要保存的图像。如果小批量的Tensor张量, 调用 `make_grid` 把Tensor张量存储为网格图像。
2. **kwargs** – `make_grid`的其他参数

译者署名

用户名	头像	职能	签名
Song		翻译	人生总要追求点什么