import torch.nn 和 import torch.nn.functional as F 使用的区别？？

1.torch.nn.Conv2d 是一个类；torch.nn.functional.conv2d 是一个函数。
换言之：
nn.Module 实现的 layer 是由 class Layer(nn.Module) 定义的特殊类
nn.functional 中的函数更像是纯函数，由 def function(input) 定义
2.两者的调用方式不同：调用 nn.xxx 时要先在里面传入超参数，然后再将数据以函数调用的方式传入 nn.xxx

```
# torch.nn
inputs =    torch.randn(64, 3, 244, 244)
self.conv = nn.Conv2d(in_channels=3, out_channels=64, kernel_size=3, padding=1)
outputs = self.conv(inputs)

# torch.nn.functional     需要同时传入数据和 weight，bias 等参数
inputs =    torch.randn(64, 3, 244, 244)
weight = torch.randn(64, 3, 3, 3)
bias = torch.randn(64)
outputs = nn.functinoal.conv2d(inputs, weight, bias, padding=1)
```

3..xxx 能够放在 nn.Sequential 里，而 nn.functional.xxx 就不行
4.nn.functional.xxx 需要自己定义 weight，每次调用时都需要手动传入 weight，而 nn.xxx 则不用
5.在使用 Dropout 时，推荐使用 nn.xxx。因为一般只有训练时才使用 Dropout，在验证或测试时不需要使用 Dropout。使用 nn.Dropout 时，如果调用 model.eval() ，模型的 Dropout 层都会关闭；但如果使用 nn.functional.dropout，在调用 model.eval() 时，不会关闭 Dropout。
6.当我们想要自定义卷积核时，是不能使用 torch.nn.ConvNd 的，因为它里面的权重都是需要学习的参数，没有办法自行定义。但是，我们可以使用 torch.nn.functional.conv2d()

2.torch.nn 具体内容？
torch.nn.functional 的函数

Convolution 函数
Pooling 函数
非线性激活函数
Normalization 函数
线性函数
Dropout 函数
距离函数（Distance functions）
损失函数（Loss functions）
Vision functions）
Convolution 函数
torch.nn.functional.conv1d(input, weight, bias=None, stride=1, padding=0, dilation=1, groups=1)
对由几个输入平面组成的输入信号应用一维卷积。

详细信息和输出形状，查看 Conv1d

参数：

input – 输入张量的形状 (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
例子：

```
>>> 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)
在由几个输入平面组成的输入图像上应用 2D 卷积。

有关详细信息和输出形状，查看 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(外通道)。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
例子：

```
>>> # With square kernels and equal stride
>>> filters = autograd.Variable(torch.randn(8,4,3,3))
>>> inputs = autograd.Variable(torch.randn(1,4,5,5))
>>> F.conv2d(inputs, filters, padding=1)
```

torch.nn.functional.conv3d(input, weight, bias=None, stride=1, padding=0, dilation=1, groups=1)
在由几个输入平面组成的输入图像上应用 3D 卷积。

有关详细信息和输出形状， 查看 Conv3d。

参数：

input – 输入张量的形状 (minibatch x in_channels x iH x iW)
weight – 过滤器的形状 (out_channels, in_channels/groups, kH, kW)

bias – 可选偏差的形状(外通道)
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 – 在输入中隐式的零填充。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
例子：

```
>>> filters = autograd.Variable(torch.randn(33, 16, 3, 3, 3))
>>> inputs = autograd.Variable(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)
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 – 输入张量的形状 (minibatch x in_channels x iW)
weight – 过滤器的形状 (in_channels x out_channels x kW)
bias – 可选偏差的形状(外通道)
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
output_padding – 在输入中隐式的零填充。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
torch.nn.functional.conv_transpose2d(input, weight, bias=None, stride=1, padding=0, output_padding=0, groups=1, dilation=1)
在由几个输入平面组成的输入图像上应用二维转置卷积，有时也称为“去卷积”。

有关详细信息和输出形状，查看 ConvTranspose2d。

参数：

input – 输入张量的形状 (minibatch x in_channels x iH x iW)
weight – 过滤器的形状 (in_channels x out_channels x kH x kW)
bias – 可选偏差的形状(外通道)
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
output_padding – 在输入中隐式的零填充。可以是单个数字或元组。默认值:0
padding – 在输入中隐式的零填充，可以是一个数字或一个元组(padh x padw)。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
dilation - 内核元素之间的间距。默认值：1
torch.nn.functional.conv_transpose3d(input, weight, bias=None, stride=1, padding=0, output_padding=0, groups=1, dilation=1)
在由几个输入平面组成的输入图像上应用三维转置卷积，有时也称为“去卷积”。

有关详细信息和输出形状，参考 ConvTranspose3d。

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 (in_channels x out_channels x kH x kW)
bias – 可选偏差的形状(外通道)
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
output_padding – 在输入中隐式的零填充。可以是单个数字或元组。默认值:0
padding – 在输入中隐式的零填充，可以是一个数字或一个元组(padh x padw)。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
dilation - 内核元素之间的间距。默认值：1
Pooling 函数
torch.nn.functional.avg_pool1d(input, kernel_size, stride=None, padding=0, ceil_mode=False, count_include_pad=True)
对由几个输入平面组成的输入信号进行一维平均池化。

有关详细信息和输出形状，参考 AvgPool1d。

参数：

kernel_size – 窗口的大小
stride – 窗口的步长。默认值为 kernel_size
padding – 在两边添加隐式零填充
ceil_mode – 当为 True 时，将使用 ceil 代替 floor 来计算输出形状
count_include_pad – 当为 True 时，将包括平均计算中的零填充。默认值：True
例子：

```
>>> # pool of square window of size=3, stride=2
>>> input = Variable(torch.Tensor([[[1,2,3,4,5,6,7]]]))
>>> F.avg_pool1d(input, kernel_size=3, stride=2)
Variable containing:
(0 ,.,.) =
   2   4   6
[torch.FloatTensor of size 1x1x3]
```

torch.nn.functional.avg_pool2d(input, kernel_size, stride=None, padding=0, ceil_mode=False, count_include_pad=True)
通过步长 dh x dw 步骤在 kh x 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)
通过步长 dt x dh x dw 步骤在 kt x kh x kw 区域中应用 3D 平均池操作。输出功能的数量等于输入平面数/ dt。

torch.nn.functional.max_pool1d(input, kernel_size, stride=None, padding=0, dilation=1, ceil_mode=False, return_indices=False)
torch.nn.functional.max_pool2d(input, kernel_size, stride=None, padding=0, dilation=1, ceil_mode=False, return_indices=False)
torch.nn.functional.max_pool3d(input, kernel_size, stride=None, padding=0, dilation=1, ceil_mode=False, return_indices=False)
torch.nn.functional.max_unpool1d(input, indices, kernel_size, stride=None, padding=0, output_size=None)
torch.nn.functional.max_unpool2d(input, indices, kernel_size, stride=None, padding=0, output_size=None)
torch.nn.functional.max_unpool3d(input, indices, kernel_size, stride=None, padding=0, output_size=None)
torch.nn.functional.lp_pool2d(input, norm_type, kernel_size, stride=None, ceil_mode=False)
torch.nn.functional.adaptive_max_pool1d(input, output_size, return_indices=False)
在由几个输入平面组成的输入信号上应用 1D 自适应最大池化。

有关详细信息和输出形状，参考 AdaptiveMaxPool1d。

参数：

output_size – 目标输出大小（单个整数）
return_indices – 是否返回池索引。默认值：False
torch.nn.functional.adaptive_max_pool2d(input, output_size, return_indices=False)
在由几个输入平面组成的输入信号上应用 2D 自适应最大池化。

有关详细信息和输出形状，参考 AdaptiveMaxPool2d。

参数：

output_size – 目标输出大小（单整数或双整数元组）
return_indices – 是否返回池索引。默认值：False
torch.nn.functional.adaptive_avg_pool1d(input, output_size)
在由几个输入平面组成的输入信号上应用 1D 自适应平均池化。

有关详细信息和输出形状，参考 AdaptiveAvgPool1d。

参数：

output_size – 目标输出大小（单整数）
torch.nn.functional.adaptive_avg_pool2d(input, output_size)
在由几个输入平面组成的输入信号上应用 2D 自适应平均池化。

有关详细信息和输出形状，参考 AdaptiveAvgPool2d。

参数：

output_size – 目标输出大小（单整数或双整数元组）
非线性激活函数
torch.nn.functional.threshold(input, threshold, value, inplace=False)
torch.nn.functional.relu(input, inplace=False)
torch.nn.functional.hardtanh(input, min_val=-1.0, max_val=1.0, inplace=False)
torch.nn.functional.relu6(input, inplace=False)
torch.nn.functional.elu(input, alpha=1.0, inplace=False)
torch.nn.functional.leaky_relu(input, negative_slope=0.01, inplace=False)
torch.nn.functional.prelu(input, weight)
torch.nn.functional.rrelu(input, lower=0.125, upper=0.3333333333333333, training=False, inplace=False)
torch.nn.functional.logsigmoid(input)
torch.nn.functional.hardshrink(input, lambd=0.5)
torch.nn.functional.tanhshrink(input)
torch.nn.functional.softsign(input)
torch.nn.functional.softplus(input, beta=1, threshold=20)
torch.nn.functional.softmin(input)
torch.nn.functional.softmax(input)
torch.nn.functional.softshrink(input, lambd=0.5)
torch.nn.functional.log_softmax(input)
torch.nn.functional.tanh(input)
torch.nn.functional.sigmoid(input)
Normalization 函数
torch.nn.functional.batch_norm(input, running_mean, running_var, weight=None, bias=None, training=False, momentum=0.1, eps=1e-05)
在指定维度上执行输入的（L_p \）归一化。 请问： [v = \frac{v}{\max(\lVert v \rVert_p, \epsilon)}]

对于每个子感应器 v 超过输入的尺寸变暗。每个子传感器被平坦化为向量，即\（\ lVert v \ rVert_p \）不是矩阵范数。

使用默认参数，通过欧几里德范数对第二维进行规范化。

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)
Dropout 函数
torch.nn.functional.dropout(input, p=0.5, training=False, inplace=False)
torch.nn.functional.alpha_dropout(input, p=0.5, training=False)
将 Alpha 退出应用于输入。 详情查看 AlphaDropout。

torch.nn.functional.dropout2d(input, p=0.5, training=False, inplace=False)
torch.nn.functional.dropout3d(input, p=0.5, training=False, inplace=False)
距离函数（Distance functions）
torch.nn.functional.pairwise_distance(x1, x2, p=2, eps=1e-06)
计算向量 v1、v2 之间的距离（成次或者成对，意思是可以计算多个，可以参看后面的参数）

\22323.png

参数：

x1 - 第一个输入的张量,
x2 - 第二个输入的张量
p - 矩阵范数的维度。默认值是 2，即二范数。
eps(float，可选) - 小值以避免除以零。默认值：1e-8
规格：

输入 - (N,D)其中 D 等于向量的维度
输出 - (N,1)其中 1 位于 dim。
例子：

>>> input1 = autograd.Variable(torch.randn(100, 128))
>>> input2 = autograd.Variable(torch.randn(100, 128))
>>> output = F.pairwise_distance(input1, input2, p=2)
>>> output.backward()
torch.nn.functional.cosine_similarity(x1, x2, dim=1, eps=1e-08)
计算向量 v1、v2 之间的距离（成次或者成对，意思是可以计算多个，可以参看后面的参数）
Distance functions

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：
输入: (* 1,D,* 2)(* 1,D,* 2) D 点定位
输出: (* 1,* 2)(* 1,* 2) 1 点定位
>>> 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（Loss functions）
torch.nn.functional.nll_loss(input, target, weight=None, size_average=True)
可能损失负对数,详细请看 NLLLoss.

参数：

input - (N,C) C=类别的个数或者在例子 2D-loss 中的(N, C, H, W)
target - (N) 其大小是 0 <= targets[i] <= C-1 1. weight (Variable, optional) – 每个类都有一个手动的重新分配的重量。如果给定，则必须是一个大小为“nclasses”的变量
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, optional) - 指定一个被忽略的目标值，并且不影响输入梯度。当 size 平均值为真时，在非被忽略的目标上的损失是平均的。
例子：

>>> # input is of size nBatch x nClasses = 3 x 5
>>> input = autograd.Variable(torch.randn(3, 5))
>>> # each element in target has to have 0 <= value < nclasses
>>> target = autograd.Variable(torch.LongTensor([1, 0, 4]))
>>> output = F.nll_loss(F.log_softmax(input), target)
>>> output.backward()
torch.nn.functional.poisson_nll_loss(input, target, log_input=True, full=False, size_average=True)
负的 log likelihood 损失函数. 详细请看 NLLLoss.

参数说明:

input – 对 Poisson 的潜在分配的预期值。
target – 随机样本目标目标 target~ Pois(input).
log_input – 如果 True，则将损失计算为 exp（input） - target input，如果 False 则输入损失 - target log（input）。默认值：True
full – 是否计算全部损失，即添加斯特林近似项。默认值：False target log（target） - target + 0.5 log（2 pi target）。
size_average – 默认情况下，每个小型服务器的损失是平均的。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
torch.nn.functional.cosine_embedding_loss(input1, input2, target, margin=0, size_average=True)
torch.nn.functional.kl_div(input, target, size_average=True)
KL 散度损失函数，详细请看 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
torch.nn.functional.cross_entropy(input, target, weight=None, size_average=True)
此标准将 log_softmax 和 nll_loss 组合在一个函数中。详细请看 CrossEntropyLoss

参数：

input - (N,C) 其中，C 是类别的个数
target - (N) 其大小是 0 <= targets[i] <= C-1 1. weight (Variable, optional) – (N) 其大小是 0 <= targets[i] <= C-1
weight (Variable, optional) – 为每个类别提供的手动权重。如果给出，必须是大小“nclasses”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
torch.nn.functional.hinge_embedding_loss（input，target，margin = 1.0，size_average = True ）
torch.nn.functional.l1_loss(input, target, size_average=True)
torch.nn.functional.mse_loss(input, target, size_average=True)
torch.nn.functional.margin_ranking_loss(input1, input2, target, margin=0, size_average=True)
torch.nn.functional.multilabel_margin_loss(input, target, size_average=True)
torch.nn.functional.multilabel_soft_margin_loss(input, target, weight=None, size_average=True)
torch.nn.functional.multi_margin_loss(input, target, p=1, margin=1, weight=None, size_average=True)
torch.nn.functional.nll_loss(input, target, weight=None, size_average=True, ignore_index=-100)
负对数似然损失。 详情看 NLLLoss。

参数：

input - \（（N，C）\）其中 C =类的数量或（ 2D，Loss ）的情况下的（N，C，H，W） target - \（（N）\），其中每个值为 0 <= targets [i] <= 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，必须是大小变量“nclasses”
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
例子：

```
>>> # input is of size nBatch x nClasses = 3 x 5
>>> input = autograd.Variable(torch.randn(3, 5))
>>> # each element in target has to have 0 <= value < nclasses
>>> target = autograd.Variable(torch.LongTensor([1, 0, 4]))
>>> output = F.nll_loss(F.log_softmax(input), target)
>>> output.backward()
```

torch.nn.functional.binary_cross_entropy_with_logits(input, target, weight=None, size_average=True)
测量目标和输出逻辑之间二进制十进制熵的函数： 详情看 BCEWithLogitsLoss。 参数：

input - 任意形状的变量
target - 与输入形状相同的变量
weight（可变，可选） - 手动重量重量，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
torch.nn.functional.smooth_l1_loss(input, target, size_average=True)
torch.nn.functional.soft_margin_loss(input, target, size_average=True)
torch.nn.functional.binary_cross_entropy(input, target, weight=None, size_average=True)
该函数计算了输出与 target 之间的二进制交叉熵，详细请看 BCELoss

参数：

input – 变量的任意形状
target – 与输入相同形状的变量
weight (Variable, optional) – 每个类都有一个手动的重新分配的重量。如果给定，则必须是一个大小为“nclasses”的变量
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。
torch.nn.functional.smooth_l1_loss(input, target, size_average=True)
torch.nn.functional.triplet_margin_loss(anchor, positive, negative, margin=1.0, p=2, eps=1e-06, swap=False)
创建一个标准来衡量一个三元组的损失，给定一个输入张量 x1，x2，x3 和一个大于 0 的值。这用于测量样本之间的相对相似性。三元组由 A、p 和 n 组成:分别是锚、正的例子和负的例子。所有输入变量的形状应该是(N，D)(N，D)。 在 V.Balntas、V.Balntas 等人的研究中，详细地描述了“距离交换”的细节描述。 torch.nn.functional.triplet_margin_loss 参数：

anchor – 输入固定的张量
positive – 输入正面的张量
negative – 输入否定的张量
p – 规范程度. 默认: 2

eps – 小量数值避免数值问题
swap – 计算距离交换 模型：
Input: (N,D)， D = 向量的维数
Output: (N,1)
>>> input1 = autograd.Variable(torch.randn(100, 128))
>>> input2 = autograd.Variable(torch.randn(100, 128))
>>> input3 = autograd.Variable(torch.randn(100, 128))
>>> output = F.triplet_margin_loss(input1, input2, input3, p=2)
>>> output.backward()
视觉 函数（vision functions）
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.

参数说明:

input (Variable) – 输入
upscale_factor (int) – 增加空间分辨率的因子.
例子:

>>> 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)
填充张量.

目前为止,只支持 2D 和 3D 填充. Currently only 2D and 3D padding supported. 当输入为 4D Tensor 的时候,pad 应该是一个 4 元素的 tuple (pad_l, pad_r, pad_t, pad_b) ,当输入为 5D Tensor 的时候,pad 应该是一个 6 元素的 tuple (pleft, pright, ptop, pbottom, pfront, pback).

形参说明:

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.
torch.nn.functional.upsample(input, size=None, scale_factor=None, mode='nearest')
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 up 抽样的模式是:最近的，双线性的(4d)，三线性(5d)

参数说明：

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’
torch.nn.functional.upsample_nearest(input, size=None, scale_factor=None)
使用最接近的邻居的像素值来对输入进行采样。 注意:这个函数是被弃用的。使用 nn.functional。upsample 相反 目前支持的空间和容量的采样是支持的(例如，预期的输入是 4 或 5 维)。参数说明：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。

torch.nn.functional.upsample_bilinear(input, size=None, scale_factor=None)
使用双线性向上采样来扩展输入。 注意: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 upsampletri 线性来进行体积(5 维)输入。 参数说明：

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 函数
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，l 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_in,L)，输出尺度（ N,C_out,L_out）的计算方式：

$$ out(Ni, C{outj})=bias(C {outj})+\sum^{C{in}-1}{k=0}weight(C{out_j},k)\bigotimes input(N_i,k) $$

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_in,L_in)
输出: (N,C_out,Lout)
输入输出的计算方式：
$$L{out}=floor((L_{in}+2padding-dilation(kernerl_size-1)-1)/stride+1)$$

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_in,H,W)，输出尺度（N,C_out,H_out,W_out）的计算方式：

$$out(Ni, C{outj})=bias(C{outj})+\sum^{C{in}-1}{k=0}weight(C{out_j},k)\bigotimes input(N_i,k)$$

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,Wout)
$$H{out}=floor((H_{in}+2padding[0]-dilation[0](kernerl_size[0]-1)-1)/stride[0]+1)$$

$$W{out}=floor((W{in}+2padding[1]-dilation[1](kernerl_size[1]-1)-1)/stride[1]+1)$$

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：
$$out(Ni, C{outj})=bias(C{outj})+\sum^{C{in}-1}{k=0}weight(C{out_j},k)\bigotimes input(N_i,k)$$

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 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} = floor((D{in} + 2 padding[0] - dilation[0] (kernel_size[0] - 1) - 1) / stride[0] + 1)) (H{out} = floor((H{in} + 2 padding[1] - dilation[1] (kernel_size[1] - 1) - 1) / stride[1] + 1)) (W{out} = floor((W{in} + 2 padding[2] - dilation[2] (kernel_size[2] - 1) - 1) / stride[2] + 1))

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作解卷积操作，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
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) stride - 2 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作解卷积操作，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) stride[0] - 2 padding[0] + kernel_size[0] + output_padding[0]) (W{out} = (W{in} - 1) stride[1] - 2 padding[1] + kernel_size[1] + output_padding[1])

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）
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作解卷积操作，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) stride[0] - 2 padding[0] + kernel_size[0] + output_padding[0]) (H{out} = (H{in} - 1) stride[1] - 2 padding[1] + kernel_size[1] + output_padding[1]) (W{out} = (W{in} - 1) stride[2] - 2 padding[2] + kernel_size[2] + output_padding[2])

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）
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_out)的计算方式是：
$$out(N_i, Cj,k)=max^{kernel_size-1}{m=0}input(N_{i},C_j,stride*k+m)$$

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_in,L_in)
输出: (N,C_out,Lout)
$$L{out}=floor((L_{in} + 2padding - dilation(kernel_size - 1) - 1)/stride + 1$$

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：
$$out(N_i, Cj,k)=max^{kH-1}{m=0}max^{kW-1}{m=0}input(N{i},C_j,stride[0]h+m,stride[1]w+n)$$

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,Wout)
$$H{out}=floor((H_{in} + 2padding[0] - dilation[0](kernel_size[0] - 1) - 1)/stride[0] + 1$$

$$W{out}=floor((W{in} + 2padding[1] - dilation[1](kernel_size[1] - 1) - 1)/stride[1] + 1$$

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：
$$out(N_i,Cj,d,h,w)=max^{kD-1}{m=0}max^{kH-1}{m=0}max^{kW-1}{m=0}$$

$$input(N_{i},C_j,stride[0]k+d,stride[1]h+m,stride[2]*w+n)$$

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,Wout)
$$D{out}=floor((D_{in} + 2padding[0] - dilation[0](kernel_size[0] - 1) - 1)/stride[0] + 1)$$

$$H{out}=floor((H{in} + 2padding[1] - dilation[1](kernel_size[0] - 1) - 1)/stride[1] + 1)$$

$$W{out}=floor((W{in} + 2padding[2] - dilation[2](kernel_size[2] - 1) - 1)/stride[2] + 1)$$

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,Hout)
$$H{out}=(H_{in}-1)stride[0]-2padding[0]+kernel_size[0]$$
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)

$$H{out}=(H{in}-1)stride[0]-2padding[0]+kernel_size[0]$$

$$W{out}=(W{in}-1)stride[1]-2padding[1]+kernel_size[1]$$

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,Wout)
$$ \begin{aligned} D{out}=(D{in}-1)stride[0]-2padding[0]+kernel_size[0]\
H{out}=(H{in}-1)stride[1]-2padding[0]+kernel_size[1]\
W{out}=(W_{in}-1)stride[2]-2padding[2]+kernel_size[2]
\end{aligned} $$

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(Variable(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：
$$out(N_i,Cj,l)=1/k*\sum^{k}{m=0}input(N{i},C{j},stride*l+m)$$
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,Lout)
$$L{out}=floor((L_{in}+2*padding-kernel_size)/stride+1)$$

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：
$$ out(N_i,Cj,h,w)=1/(kHkW)\sum^{kH-1}{m=0}\sum^{kW-1}{n=0}input(N{i},C_{j},stride[0]h+m,stride[1]w+n)$$

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,Wout)
$$\begin{aligned} H{out}=floor((H{in}+2*padding[0]-kernel_size[0])/stride[0]+1)\
W{out}=floor((W_{in}+2*padding[1]-kernel_size[1])/stride[1]+1) \end{aligned} $$

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：

$$ \begin{aligned} out(N_i,Cj,d,h,w)=1/(kDkHkW)*\sum^{kD-1}{k=0}\sum^{kH-1}{m=0}\sum^{kW-1}{n=0}input(N{i},C{j},stride[0]d+k,stride[1]h+m,stride[2]*w+n) \end{aligned} $$ 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_in,H_in,W_in)
输出大小:(N,C,D_out,H_out,Wout) $$\begin{aligned} D{out}=floor((D{in}+2*padding[0]-kernel_size[0])/stride[0]+1)\ H{out}=floor((H{in}+2*padding[1]-kernel_size[1])/stride[1]+1)\ W{out}=floor((W_{in}+2*padding[2]-kernel_size[2])/stride[2]+1) \end{aligned} $$

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$kH*kW$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*K 的窗口），也可以是一个元组（kh*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 13x12
>>> m = nn.FractionalMaxPool2d(3, output_size=(13, 12))
>>> # pool of square window and target output size being half of input image size
>>> m = nn.FractionalMaxPool2d(3, output_ratio=(0.5, 0.5))
>>> input = autograd.Variable(torch.randn(20, 16, 50, 32))
>>> output = m(input)
```

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：
$$f(x)=pow(sum(X,p),1/p)$$

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_in,W_in)
输出：(N,C,H_out,Wout)
$$\begin{aligned} H{out} = floor((H{in}+2padding[0]-dilation[0](kernel_size[0]-1)-1)/stride[0]+1)\
W{out} = floor((W_{in}+2padding[1]-dilation[1](kernel_size[1]-1)-1)/stride[1]+1) \end{aligned} $$
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${ReLU}(x)= max(0, x)$，

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${ReLU6}(x) = min(max(0,x), 6)$，

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$PReLU(x) = max(0,x) + a * min(0,x)$，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$f(x) = max(0, x) + {negative_slope} * min(0, x)$

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：

$$ y = x ,if\ x >= threshold\ y = value,if\ x < threshold $$

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，

$$ f(x) = +1, if\ x > 1;\ f(x) = -1, if\ x < -1;\ f(x) = x, otherwise $$

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，$LogSigmoid(x) = log(1 / (1 + e^{-x}))$

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：

$$ Tanhshrink(x) = x - Tanh(x) $$

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：

$$f_i(x) = \frac{e^{(-x_i - shift)}} { \sum^j e^{(-x_j - shift)}},shift = max (x_i)$$

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：

$$f_i(x) = \frac{e^{(x_i - shift)}} { \sum^j e^{(x_j - shift)}},shift = max (x_i)$$

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

$$ y = \frac{x - mean[x]}{ \sqrt{Var[x]} + \epsilon} * gamma + beta $$

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

$$ y = \frac{x - mean[x]}{ \sqrt{Var[x]} + \epsilon} * gamma + beta $$

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

$$ y = \frac{x - mean[x]}{ \sqrt{Var[x]} + \epsilon} * gamma + beta $$

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 $$ ht=tanh(w{ih} xt+b{ih}+w_{hh} h{t-1}+b{hh}) $$ 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{split}\begin{array}{ll} it = \mathrm{sigmoid}(W{ii} xt + b{ii} + W{hi} h{(t-1)} + b_{hi}) \ ft = \mathrm{sigmoid}(W{if} xt + b{if} + W{hf} h{(t-1)} + b_{hf}) \ gt = \tanh(W{ig} xt + b{ig} + W{hc} h{(t-1)} + b_{hg}) \ ot = \mathrm{sigmoid}(W{io} xt + b{io} + W{ho} h{(t-1)} + b_{ho}) \ c_t = ft * c{(t-1)} + i_t g_t \ h_t = o_t \tanh(c_t) \end{array}\end{split}]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]](http://pytorch.org/docs/master/_modules/torch/nn/modules/rnn.html#GRU)

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:

\[\begin{split}\begin{array}{ll} r_t = \mathrm{sigmoid}(W_{ir} x_t + b_{ir} + W_{hr} h_{(t-1)} + b_{hr}) \\ z_t = \mathrm{sigmoid}(W_{iz} x_t + b_{iz} + W_{hz} h_{(t-1)} + b_{hz}) \\ n_t = \tanh(W_{in} x_t + b_{in} + r_t * (W_{hn} h_{(t-1)}+ b_{hn})) \\ h_t = (1 - z_t) * n_t + z_t * h_{(t-1)} \\ \end{array}\end{split}\]

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