

torch.nn

Parameters

class torch.nn.Parameter()

`Variable` 的一种，常被用于模块参数(`module parameter`)。

`Parameters` 是 `Variable` 的子类。`Parameters` 和 `Modules` 一起使用的时候会有一些特殊的属性，即：当 `Parameters` 赋值给 `Module` 的属性的时候，他会自动的被加入到 `Module` 的参数列表中(即：会出现在 `parameters()` 迭代器中)。将 `Variable` 赋值给 `Module` 属性则不会有这样的影响。这样做的原因是：我们有时候会需要缓存一些临时的状态(`state`)，比如：模型中 `RNN` 的最后一个隐状态。如果没有 `Parameter` 这个类的话，那么这些临时变量也会注册成为模型变量。

`Variable` 与 `Parameter` 的另一个不同之处在于，`Parameter` 不能被 `volatile` (即：无法设置 `volatile=True`)而且默认 `requires_grad=True`。`Variable` 默认 `requires_grad=False`。

参数说明:

- data (Tensor) – parameter tensor.
- requires_grad (bool, optional) – 默认为 `True`，在 `BP` 的过程中会对其求微分。

Containers (容器) :

class torch.nn.Module

所有网络的基类。

你的模型也应该继承这个类。

`Modules` 也可以包含其它 `Modules`，允许使用树结构嵌入他们。你可以将子模块赋值给模型属性。

```
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))
```

通过上面方式赋值的 `submodule` 会被注册。当调用 `.cuda()` 的时候, `submodule` 的参数也会转换为 `cuda Tensor`。

add_module(name, module)

将一个 `child module` 添加到当前 `module`。被添加的 `module` 可以通过 `name` 属性来获取。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))
```

children()

Returns an iterator over immediate children modules. 返回当前模型子模块的迭代器。

```
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.add_module("conv1", nn.Conv2d(20, 10, 4))
model = Model()

for sub_module in model.children():
    print(sub_module)
```

```
Conv2d(10, 20, kernel_size=(4, 4), stride=(1, 1))
Conv2d(20, 10, kernel_size=(4, 4), stride=(1, 1))
```

cpu(device_id=None)

 v: latest ▼

将所有的模型参数(`parameters`)和 `buffers` 复制到 `CPU`

NOTE：官方文档用的move，但我觉着 `copy` 更合理。

`cuda(device_id=None)`

将所有的模型参数(`parameters`)和 `buffers` 赋值 `GPU`

参数说明:

- `device_id` (int, optional) – 如果指定的话，所有的模型参数都会复制到指定的设备上。

`double()`

将 `parameters` 和 `buffers` 的数据类型转换成 `double`。

`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 `parameters` 和 `buffers` 的数据类型转换成 `float`。

`forward(* input)`

定义了每次执行的 计算步骤。 在所有的子类中都需要重写这个函数。

`half()`

将 `parameters` 和 `buffers` 的数据类型转换成 `half`。

`load_state_dict(state_dict)`

将 `state_dict` 中的 `parameters` 和 `buffers` 复制到此 `module` 和它的后代中。 `state_dict` 中的 `key` 必须和 `model.state_dict()` 返回的 `key` 一致。 **NOTE**：用来加载模型参数。

参数说明:

- `state_dict` (dict) – 保存 `parameters` 和 `persistent buffers` 的字典。

`modules()`

 v: latest ▼

返回一个包含 当前模型 所有模块的迭代器。

```
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.add_module("conv1", nn.Conv2d(20, 10, 4))
model = Model()

for module in model.modules():
    print(module)
```

```
Model (
  (conv): Conv2d(10, 20, kernel_size=(4, 4), stride=(1, 1))
  (conv1): Conv2d(20, 10, kernel_size=(4, 4), stride=(1, 1))
)
Conv2d(10, 20, kernel_size=(4, 4), stride=(1, 1))
Conv2d(20, 10, kernel_size=(4, 4), stride=(1, 1))
```

可以看出, `modules()` 返回的 `iterator` 不止包含子模块。这是和 `children()` 的不同。

NOTE: 重复的模块只被返回一次(`children()` 也是)。在下面的例子中, `submodule` 只会被返回一次:

```
import torch.nn as nn

class Model(nn.Module):
    def __init__(self):
        super(Model, self).__init__()
        submodule = nn.Conv2d(10, 20, 4)
        self.add_module("conv", submodule)
        self.add_module("conv1", submodule)
model = Model()

for module in model.modules():
    print(module)
```

```
Model (
  (conv): Conv2d(10, 20, kernel_size=(4, 4), stride=(1, 1))
  (conv1): Conv2d(10, 20, kernel_size=(4, 4), stride=(1, 1))
)
Conv2d(10, 20, kernel_size=(4, 4), stride=(1, 1))
```

named_children()

返回 包含 模型当前子模块 的迭代器, `yield` 模块名字和模块本身。

例子:

```
for name, module in model.named_children():
    if name in ['conv4', 'conv5']:
        print(module)
```

named_modules(memo=None, prefix="")[source]

 v: latest ▼

返回包含网络中所有模块的迭代器, `yielding` 模块名和模块本身。

注意：

重复的模块只被返回一次(`children()`也是)。在下面的例子中, `submodule` 只会被返回一次。

`parameters(memo=None)`

返回一个 包含模型所有参数 的迭代器。

一般用来当作 `optimizer` 的参数。

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 `module` 上注册一个 `backward hook`。

每次计算 `module` 的 `inputs` 的梯度的时候, 这个 `hook` 会被调用。 `hook` 应该拥有下面的 `signature`。

```
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 `persistent buffer`。

`persistent buffer` 通常被用在这么一种情况：我们需要保存一个状态, 但是这个状态不能看作为模型参数。例如：, `BatchNorm's` `running_mean` 不是一个 `parameter`, 但是它也是需要保存的状态之一。

`Buffers` 可以通过注册时候的 `name` 获取。

 v: latest ▼

NOTE :我们可以用 `buffer` 保存 `moving average`

例子：

```
self.register_buffer('running_mean', torch.zeros(num_features))

self.running_mean
```

register_forward_hook(hook)

在 `module` 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_parameter(name, param)

向 `module` 添加 `parameter`

`parameter` 可以通过注册时候的 `name` 获取。

state_dict(destination=None, prefix="")[source]

返回一个字典，保存着 `module` 的所有状态（`state`）。

`parameters` 和 `persistent buffers` 都会包含在字典中，字典的 `key` 就是 `parameter` 和 `buffer` 的 `names`。

例子：

```
import torch
from torch.autograd import Variable
import torch.nn as nn

class Model(nn.Module):
    def __init__(self):
        super(Model, self).__init__()
        self.conv2 = nn.Linear(1, 2)
        self.vari = Variable(torch.rand([1]))
        self.par = nn.Parameter(torch.rand([1]))
        self.register_buffer("buffer", torch.randn([2,3]))

model = Model()
print(model.state_dict().keys())
```

```
odict_keys(['par', 'buffer', 'conv2.weight', 'conv2.bias'])
```

 v: latest ▼

train(mode=True)

将 `module` 设置为 `training mode`。

仅仅当模型中有 `Dropout` 和 `BatchNorm` 是才会有影响。

`zero_grad()`

将 `module` 中的所有模型参数的梯度设置为0。

`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`。

为了更容易的理解如何使用 `Sequential`，下面给出了一个例子：

```
# 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)[source]`

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)` – 将要被添加到 `ModuleList` 中的 `modules` 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)[source]

等价于 list 的 `append()`

参数说明:

- module (nn.Module) – 要 append 的 `module`

extend(modules)[source]

等价于 `list` 的 `extend()` 方法

参数说明:

- modules (list) – list of modules to append

class torch.nn.ParameterList(parameters=None)

将 `submodules` 保存在一个 `list` 中。

`ParameterList` 可以像一般的 `Python list` 一样被 `索引`。而且 `ParameterList` 中包含的 `parameters` 已经被正确的注册，对所有的 `module method` 可见。

参数说明:

- modules (list, optional) – a list of nn.Parameter

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)[source]

等价于 `python list` 的 `append` 方法。

参数说明:

- parameter (nn.Parameter) – parameter to append

extend(parameters)[source]

等价于 `python list` 的 `extend` 方法。

参数说明:

- parameters (list) – list of parameters to append

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:

$$out(N_i, C_{out_j}) = bias(C_{out_j}) + \sum_{k=0}^{C_{in}-1} weight(C_{out_j}, k) \cdot input(N_i, k)$$

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
- 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:

输入: (N,C_in,L_in)

输出: (N,C_out,L_out)

输入输出的计算方式:

$$L_{out} = \text{floor}((L_{in} + 2padding - dilation(kernel_size - 1) - 1) / stride + 1)$$

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:

$$out(N_i, C_{out_j}) = bias(C_{out_j}) + \sum^{C_{in}-1}_{k=0} weight(C_{out_j}, k) \otimes input(N_i, k)$$

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` 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`)

example:

```
>>> # 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:

$$out(N_i, C_{out_j}) = bias(C_{out_j}) + \sum_{k=0}^{C_{in}-1} weight(C_{out_j}, k) \cdot input(N_i, k)$$

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

 v: latest ▼

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)

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

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

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

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`)

example:

```
>>> # 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)`

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) 。

 v: latest ▾

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)

$L_{out} = (L_{in} - 1) \times stride - 2 \times padding + kernel_size + output_padding$

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)`

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:  [v: latest](#) ▼
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)

$H_{out} = (H_{in} - 1) \cdot stride[0] - 2 \cdot padding[0] + kernel_size[0] + output_padding[0]$

$W_{out} = (W_{in} - 1) \cdot stride[1] - 2 \cdot padding[1] + kernel_size[1] + output_padding[1]$

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])
```

`torch.nn.ConvTranspose3d(in_channels, out_channels, kernel_size, stride=1, padding=0, output_padding=0, groups=1, bias=True)`

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

 v: latest ▼

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,H_in, W_in)

输出: (N,C_out,H_out,W_out)

$$D_{out} = (D_{in} - 1) \cdot stride[0] - 2 \cdot padding[0] + kernel_size[0] + output_padding[0]$$

$$H_{out} = (H_{in} - 1) \cdot stride[1] - 2 \cdot padding[1] + kernel_size[1] + output_padding[0]$$

$$W_{out} = (W_{in} - 1) \cdot stride[2] - 2 \cdot padding[2] + kernel_size[2] + output_padding[2]$$

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：

$$out(N_i, C_j, k) = \max_{m=0}^{kernel_size-1} input(N_i, C_j, stride*k+m)$$

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，代替默认的下取整的操作

shape:

输入: (N,C_in,L_in)

输出: (N,C_out,L_out)

$$L_{out} = \text{floor}((L_{in} + 2padding - dilation(kernel_size - 1) - 1)/stride + 1)$$

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)

 v: latest ▼

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：

$$out(N_i, C_j, d, h, w) = \max_{m=0}^{kD-1} \max_{n=0}^{kH-1} \max_{p=0}^{kW-1} input(N_i, C_j, d, h, w)$$

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

如果 `padding` 不是0, 会在输入的每一边添加相应数目0

`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, 代替默认的下取整的操作

shape:

输入: (N,C,H_{in},W_{in})

输出: (N,C,H_{out},W_{out})

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

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

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

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)

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


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  [v: latest](#)

`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)
$$D_{out} = (D_{in} - 1) \cdot stride[0] - 2 \cdot padding[0] + kernel_size[0]$$
$$H_{out} = (H_{in} - 1) \cdot stride[1] - 2 \cdot padding[1] + kernel_size[1]$$
$$W_{out} = (W_{in} - 1) \cdot stride[2] - 2 \cdot padding[2] + kernel_size[2]$$

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(1, 1, 20, 16, 51, 33, 15))
>>> unpooled_output = unpool(output, indices)
>>> unpooled_output.size()
torch.Size([1, 1, 20, 16, 51, 33, 15])
```

class torch.nn.AvgPool1d(kernel_size, stride=None, padding=0, ceil_mode=False, count_include_pad=True)

 v: latest ▼

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：

$$out(N_i, C_j, l) = 1/k \sum_{m=0}^{k-1} input(N_i, C_j, stride_l + m)$$

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(torch.nn.Variable(torch.FloatTensor([[[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：

$$out(N_i, C_j, h, w) = 1/(kH kW) \sum_{m=0}^{kH-1} \sum_{n=0}^{kW-1} input(N_i, C_j, stride[0]h+m, stride[1]w+n)$$

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`  `v: latest` ▼
- `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:

$$\begin{aligned} out(N_i, C_j, d, h, w) &= 1 / (kD kH kW) \sum_{k=0}^{kD-1} \sum_{h=0}^{kH-1} \sum_{w=0}^{kW-1} input(N_i, C_j, stride[0]d+k, stride[1]h+m, stride[2]w+n) \end{aligned}$$

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,W_out)
$$\begin{aligned} D_{out} &= \text{floor}((D_{in} + 2padding[0] - kernel_size[0]) / stride[0] + 1) \\ H_{out} &= \text{floor}((H_{in} + 2padding[1] - kernel_size[1]) / stride[1] + 1) \\ W_{out} &= \text{floor}((W_{in} + 2padding[2] - kernel_size[2]) / stride[2] + 1) \end{aligned}$$

$$\end{aligned}$$

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：

$$f(x) = \text{pow}(\text{sum}(X, p), 1/p)$$

- 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_{in},W_{in})
- 输出: (N,C,H_{out},W_{out})
$$H_{out} = \text{floor}((H_{in} + 2padding[0] - dilation[0]) / (kernel_size[0] - 1) / stride[0] + 1)$$
$$W_{out} = \text{floor}((W_{in} + 2padding[1] - dilation[1]) / (kernel_size[1] - 1) / stride[1] + 1)$$

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 [\[source\]](#)

`class torch.nn.ReLU(inplace=False)` [\[source\]](#)

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)` [\[source\]](#)

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)` [\[source\]](#)

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)` [\[source\]](#)

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)` [\[source\]](#)

对输入的每一个元素运用 $f(x) = \max(0, x) + \{\text{negative_slope}\} * \min(0, x)$

参数:

 v: latest ▼

- 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)` [\[source\]](#)

Threshold定义:

$$y = x, \text{ if } x \geq \text{threshold} \quad y = \text{value}, \text{ if } x < \text{threshold}$$

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)` [\[source\]](#)

对每个元素,

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

线性区域的范围 $[-1, 1]$ 可以被调整

 v: latest ▼

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` [\[source\]](#)

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` [\[source\]](#)

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

 [v: latest](#) ▼

例子:

```
>>> m = nn.Tanh()
>>> input = autograd.Variable(torch.randn(2))
>>> print(input)
>>> print(m(input))
```

class torch.nn.LogSigmoid [\[source\]](#)

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) [\[source\]](#)

对每个元素运用Softplus函数, Softplus 定义如下:

$$f(x) = \frac{1}{\text{beta}} * \log(1 + e^{(\text{beta} * x_i)})$$

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)`[\[source\]](#)

对每个元素运用Softshrink函数，Softshrink函数定义如下：

$$f(x) = x - \lambda, \text{ if } x > \lambda \quad f(x) = x + \lambda, \text{ if } x < -\lambda \quad f(x) = 0, \text{ otherwise}$$

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` [\[source\]](#)

$$f(x) = x / (1 + |x|)$$

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)`[\[source\]](#)

对每个元素运用Tanhshrink函数，Tanhshrink函数定义如下：

 v: latest ▼

$$\text{Tanhshrink}(x) = x - \text{Tanh}(x)$$

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` [\[source\]](#)

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:

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

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` [\[source\]](#)

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` [\[source\]](#)

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 [\[source\]](#)

`class torch.nn.BatchNorm1d(num_features, eps=1e-05, momentum=0.1, affine=True)` [\[source\]](#)

对小批量(mini-batch)的2d或3d输入进行批标准化(Batch Normalization)操作

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

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)[\[source\]](#)

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

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

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)`[\[source\]](#)

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

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

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)
```

class torch.nn.RNN(args, * kwargs)[source]

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 `bih` 和 `bhh`，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)[source]

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:
$$\begin{aligned} i_t &= \text{sigmoid}(W_{ii}x_t + b_{ii} + W_{hi}h_{t-1} + b_{hi}) \\ f_t &= \text{sigmoid}(W_{if}x_t + b_{if} + W_{hf}h_{t-1} + b_{hf}) \\ o_t &= \text{sigmoid}(W_{io}x_t + b_{io} + W_{ho}h_{t-1} + b_{ho}) \\ g_t &= \tanh(W_{ig}x_t + b_{ig} + W_{hg}h_{t-1} + b_{hg}) \\ c_t &= f_{t-1} + i_t g_t \\ h_t &= o_t * \tanh(c_t) \end{aligned}$$
 h_t 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, 细胞和输出门。

参数说明:

- input_size – 输入的特征维度
- hidden_size – 隐状态的特征维度
- num_layers – 层数 (和时序展开要区分开)
- bias – 如果为 `False`, 那么 `LSTM` 将不会使用 b_{ih}, b_{hh} , 默认为 `True`。
- batch_first – 如果为 `True`, 那么输入和输出 `Tensor` 的形状为 `(batch, seq, feature)`
- dropout – 如果非零的话, 将会在 `RNN` 的输出上加个 `dropout`, 最后一层除外。
- bidirectional – 如果为 `True`, 将会变成一个双向 `RNN`, 默认为 `False`。

`LSTM` 输入: input, (h_0, c_0)

- input (seq_len, batch, input_size): 包含输入序列特征的 `Tensor`。也可以是 `packed variable`, 详见 `[pack_padded_sequence](#torch.nn.utils.rnn.pack_padded_sequence(input, lengths, batch_first=False[source]))`
- h_0 (num_layers * num_directions, batch, hidden_size): 保存着 `batch` 中每个元素的初始化隐状态的 `Tensor`
- c_0 (num_layers * num_directions, batch, hidden_size): 保存着 `batch` 中每个元素的初始化细胞状态的 `Tensor`

`LSTM` 输出 output, (h_n, c_n)

- output (seq_len, batch, hidden_size * num_directions): 保存 `RNN` 最后一层的输出的 `Tensor`。如果输入是 `torch.nn.utils.rnn.PackedSequence`, 那么输出也是 `torch.nn.utils.rnn.PackedSequence`。
- h_n (num_layers * num_directions, batch, hidden_size): `Tensor`, 保存着 `RNN` 最后一步的隐状态。

- `c_n(num_layers * num_directions, batch, hidden_size)`: `Tensor`，保存着 `RNN` 最后一个时间步的细胞状态。

`LSTM` 模型参数:

- `weight_ih_l[k]` - 第 `k` 层可学习的 `input-hidden` 权重($W_{ii}|W_{if}|W_{ig}|W_{io}$)，形状为 `(input_size x 4*hidden_size)`
- `weight_hh_l[k]` - 第 `k` 层可学习的 `hidden-hidden` 权重($W_{hi}|W_{hf}|W_{hg}|W_{ho}$)，形状为 `(hidden_size x 4*hidden_size)`。
- `bias_ih_l[k]` - 第 `k` 层可学习的 `input-hidden` 偏置($b_{ii}|b_{if}|b_{ig}|b_{io}$)，形状为 `(4*hidden_size)`
- `bias_hh_l[k]` - 第 `k` 层可学习的 `hidden-hidden` 偏置($b_{hi}|b_{hf}|b_{hg}|b_{ho}$)，形状为 `(4*hidden_size)`。 示例:

```
lstm = nn.LSTM(10, 20, 2)
input = Variable(torch.randn(5, 3, 10))
h0 = Variable(torch.randn(2, 3, 20))
c0 = Variable(torch.randn(2, 3, 20))
output, hn = lstm(input, (h0, c0))
```

`class torch.nn.GRU(args, * kwargs)[source]`

将一个多层的 `GRU` 用于输入序列。

对输入序列中的每个元素，每层进行了一下计算：

$$\begin{aligned} r_t &= \text{sigmoid}(W_{ir}x_t + b_{ir} + W_{hr}h_{(t-1)} + b_{hr}) \\ i_t &= \text{sigmoid}(W_{ii}x_t + b_{ii} + W_{hi}h_{(t-1)} + b_{hi}) \\ n_t &= \tanh(W_{in}x_t + b_{in} + r_t(W_{hn}h_{(t-1)} + b_{hn})) \\ h_t &= (1 - i_t)n_t + i_t h_{(t-1)} \end{aligned}$$

h_t 是时间 t 上的隐状态， x_t 是前一层 t 时刻的隐状态或者是第一层的 t 时刻的输入， r_t, i_t, n_t 分别是重置门，输入门和新门。

参数说明： - `input_size` - 期望的输入 x 的特征值的维度 - `hidden_size` - 隐状态的维度 - `num_layers` - `RNN` 的层数。 - `bias` - 如果为 `False`，那么 `RNN` 层将不会使用 `bias`，默认为 `True`。 - `batch_first` - 如果为 `True` 的话，那么输入和输出的 `tensor` 的形状是 `(batch, seq, feature)`。 - `dropout` - 如果非零的话，将会在 `RNN` 的输出上加个 `dropout`，最后一层除外。 - `bidirectional` - 如果为 `True`，将会变成一个双向 `RNN`，默认为 `False`。

输入： `input, h_0`

- `input(seq_len, batch, input_size)`: 包含输入序列特征的 `Tensor`。也可以是 `packed variable`，详见 `[pack_padded_sequence](#torch.nn.utils.rnn.pack_padded_sequence(input, lengths, batch_first=False[source]))`。
- `h_0(num_layers * num_directions, batch, hidden_size)`: 保存着 `batch` 中每个元素的初始化隐状态的 `Tensor`

输出: output, h_n

- output (seq_len, batch, hidden_size * num_directions): 保存 RNN 最后一层的输出的 Tensor。如果输入是 torch.nn.utils.rnn.PackedSequence, 那么输出也是 torch.nn.utils.rnn.PackedSequence。
- h_n (num_layers * num_directions, batch, hidden_size): Tensor, 保存着 RNN 最后一个时间步的隐状态。

变量:

- weight_ih_l[k] - 第 k 层可学习的 input-hidden 权重 ($W_{ir}|W_{ii}|W_{in}$), 形状为 (input_size x 3*hidden_size)。
- weight_hh_l[k] - 第 k 层可学习的 hidden-hidden 权重 ($W_{hr}|W_{hi}|W_{hn}$), 形状为 (hidden_size x 3*hidden_size)。
- bias_ih_l[k] - 第 k 层可学习的 input-hidden 偏置 ($b_{ir}|b_{ii}|b_{in}$), 形状为 (3*hidden_size)。
- bias_hh_l[k] - 第 k 层可学习的 hidden-hidden 偏置 ($b_{hr}|b_{hi}|b_{hn}$), 形状为 (3*hidden_size)。

例子:

```
rnn = nn.GRU(10, 20, 2)
input = Variable(torch.randn(5, 3, 10))
h0 = Variable(torch.randn(2, 3, 20))
output, hn = rnn(input, h0)
```

class torch.nn.RNNCell(input_size, hidden_size, bias=True, nonlinearity='tanh')[source]

一个 Elman RNN cell, 激活函数是 tanh 或 ReLU, 用于输入序列。将一个多层的 Elman RNNCell, 激活函数为 tanh 或者 ReLU, 用于输入序列。 $h' = \tanh(w_{ih}x + b_{ih} + w_{hh}h + b_{hh})$ 如果 nonlinearity=relu, 那么将会使用 ReLU 来代替 tanh。

参数:

- input_size - 输入 x , 特征的维度。
- hidden_size - 隐状态特征的维度。
- bias - 如果为 False, RNN cell 中将不会加入 bias, 默认为 True。
- nonlinearity - 用于选择非线性激活函数 [tanh | relu]. 默认值为: tanh

输入: input, hidden

 v: latest ▼

- input (batch, input_size): 包含输入特征的 tensor。
- hidden (batch, hidden_size): 保存着初始隐状态值的 tensor。

输出: h'

- h' (batch, hidden_size): 下一个时刻的隐状态。

变量:

- weight_ih – input-hidden 权重, 可学习, 形状是 (input_size x hidden_size)。
- weight_hh – hidden-hidden 权重, 可学习, 形状是 (hidden_size x hidden_size)
- bias_ih – input-hidden 偏置, 可学习, 形状是 (hidden_size)
- bias_hh – hidden-hidden 偏置, 可学习, 形状是 (hidden_size)

例子:

```
rnn = nn.RNNCell(10, 20)
input = Variable(torch.randn(6, 3, 10))
hx = Variable(torch.randn(3, 20))
output = []
for i in range(6):
    hx = rnn(input[i], hx)
    output.append(hx)
```

class torch.nn.LSTMCell(input_size, hidden_size, bias=True)[source]

LSTM cell。
$$i = \text{sigmoid}(W_{ii}x + b_{ii} + W_{hi}h + b_{hi})$$

$$f = \text{sigmoid}(W_{if}x + b_{if} + W_{hf}h + b_{hf})$$

$$o = \text{sigmoid}(W_{io}x + b_{io} + W_{ho}h + b_{ho})$$

$$g = \text{tanh}(W_{ig}x + b_{ig} + W_{hg}h + b_{hg})$$

$$c' = f_{t-1} + i_t g$$

$$h' = o_t * \text{tanh}(c')$$

参数:

- input_size – 输入的特征维度。
- hidden_size – 隐状态的维度。
- bias – 如果为 `False`, 那么将不会使用 `bias`。默认为 `True`。

LSTM 输入: input, (h_0, c_0)

- input (seq_len, batch, input_size): 包含输入序列特征的 `Tensor`。也可以是 `packed variable`, 详见 `[pack_padded_sequence](#torch.nn.utils.rnn.pack_padded_sequence(input, lengths, batch_first=False[source])`
- h_0 (batch, hidden_size): 保存着 `batch` 中每个元素的初始化隐状态的 `Tensor`
- c_0 (batch, hidden_size): 保存着 `batch` 中每个元素的初始化细胞状态的 `Tensor`

输出: h_1, c_1

- h_1 (batch, hidden_size): 下一个时刻的隐状态。
- c_1 (batch, hidden_size): 下一个时刻的细胞状态。

LSTM 模型参数:

- weight_ih – input-hidden 权重($W_{ii}|W_{if}|W_{ig}|W_{io}$), 形状为 $(input_size \times 4 \times hidden_size)$
- weight_hh – hidden-hidden 权重($W_{hi}|W_{hf}|W_{hg}|W_{ho}$), 形状为 $(hidden_size \times 4 \times hidden_size)$ 。
- bias_ih – input-hidden 偏置($b_{ii}|b_{if}|b_{ig}|b_{io}$), 形状为 $(4 \times hidden_size)$
- bias_hh – hidden-hidden 偏置($b_{hi}|b_{hf}|b_{hg}|b_{ho}$), 形状为 $(4 \times hidden_size)$ 。

Examples:

```
rnn = nn.LSTMCell(10, 20)
input = Variable(torch.randn(6, 3, 10))
hx = Variable(torch.randn(3, 20))
cx = Variable(torch.randn(3, 20))
output = []
for i in range(6):
    hx, cx = rnn(input[i], (hx, cx))
    output.append(hx)
```

class torch.nn.GRUCell(input_size, hidden_size, bias=True)[source]

一个 GRU cell。
$$r = \text{sigmoid}(W_{ir}x + b_{ir} + W_{hr}h + b_{hr})$$
$$i = \text{sigmoid}(W_{ii}x + b_{ii} + W_{hi}h + b_{hi})$$
$$n = \tanh(W_{in}x + b_{in} + r(W_{hn}h + b_{hn}))$$
$$h' = (1-i)n + i*h$$

参数说明: - input_size – 期望的输入 x 的特征值的维度 - hidden_size – 隐状态的维度 - bias – 如果为 `False`, 那么 RNN 层将不会使用 `bias`, 默认为 `True`。

输入: input, h_0

- input (batch, input_size): 包含输入特征的 Tensor
- h_0 (batch, hidden_size): 保存着 batch 中每个元素的初始化隐状态的 Tensor

输出: h_1

- h_1 (batch, hidden_size): Tensor, 保存着 RNN 下一个时刻的隐状态。

变量:

- weight_ih – input-hidden 权重($W_{ir}|W_{ii}|W_{in}$), 形状为 $(input_size \times 3 \times hidden_size)$
- weight_hh – hidden-hidden 权重($W_{hr}|W_{hi}|W_{hn}$), 形状为 $(hidden_size \times 3 \times hidden_size)$ 。
- bias_ih – input-hidden 偏置($b_{ir}|b_{ii}|b_{in}$), 形状为 $(3 \times hidden_size)$
- bias_hh – hidden-hidden 偏置($b_{hr}|b_{hi}|b_{hn}$), 形状为 $(3 \times hidden_size)$ 。

例子：

```
rnn = nn.GRUCell(10, 20)
input = Variable(torch.randn(6, 3, 10))
hx = Variable(torch.randn(3, 20))
output = []
for i in range(6):
    hx = rnn(input[i], hx)
    output.append(hx)
```

Linear layers

```
class torch.nn.Linear(in_features, out_features, bias=True)
```

对输入数据做线性变换： $y = Ax + b$

参数：

- **in_features** - 每个输入样本的大小
- **out_features** - 每个输出样本的大小
- **bias** - 若设置为False，这层不会学习偏置。默认值：True

形状：

- **输入：** $(N, in_features)$
- **输出：** $(N, out_features)$

变量：

- **weight** - 形状为 $(out_features \times in_features)$ 的模块中可学习的权值
- **bias** - 形状为 $(out_features)$ 的模块中可学习的偏置

例子：

```
>>> m = nn.Linear(20, 30)
>>> input = autograd.Variable(torch.randn(128, 20))
>>> output = m(input)
>>> print(output.size())
```

Dropout layers

```
class torch.nn.Dropout(p=0.5, inplace=False)
```

随机将输入张量中部分元素设置为0。对于每次前向调用，被置0的元素都是随机的。

 v: latest ▼

参数：

- `p` - 将元素置0的概率。默认值：0.5
- `in-place` - 若设置为True，会在原地执行操作。默认值：False

形状：

- **输入：** 任意。输入可以为任意形状。
- **输出：** 相同。输出和输入形状相同。

例子：

```
>>> m = nn.Dropout(p=0.2)
>>> input = autograd.Variable(torch.randn(20, 16))
>>> output = m(input)
```

```
class torch.nn.Dropout2d(p=0.5, inplace=False)
```

随机将输入张量中整个通道设置为0。对于每次前向调用，被置0的通道都是随机的。

通常输入来自Conv2d模块。

像在论文[Efficient Object Localization Using Convolutional Networks](#)，如果特征图中相邻像素是强相关的（在前几层卷积层很常见），那么iid dropout不会归一化激活，而只会降低学习率。

在这种情形，`nn.Dropout2d()`可以提高特征图之间的独立程度，所以应该使用它。

参数：

- `p(float, optional)` - 将元素置0的概率。
- `in-place(bool, optional)` - 若设置为True，会在原地执行操作。

形状：

- **输入：** (N, C, H, W)
- **输出：** (N, C, H, W) （与输入形状相同）

例子：

```
>>> m = nn.Dropout2d(p=0.2)
>>> input = autograd.Variable(torch.randn(20, 16, 32, 32))
>>> output = m(input)
```

```
class torch.nn.Dropout3d(p=0.5, inplace=False)
```

 v: latest ▼

随机将输入张量中整个通道设置为0。对于每次前向调用，被置0的通道都是随机的。

通常输入来自Conv3d模块。

像在论文[Efficient Object Localization Using Convolutional Networks](#)，如果特征图中相邻像素是强相关的（在前几层卷积层很常见），那么iid dropout不会归一化激活，而只会降低学习率。

在这种情形，`nn.Dropout3d()`可以提高特征图之间的独立程度，所以应该使用它。

参数：

- `p(float, optional)` - 将元素置0的概率。
- `in-place(bool, optional)` - 若设置为True，会在原地执行操作。

形状：

- 输入： N, C, D, H, W
- 输出： (N, C, D, H, W) （与输入形状相同）

例子：

```
>>> m = nn.Dropout3d(p=0.2)
>>> input = autograd.Variable(torch.randn(20, 16, 4, 32, 32))
>>> output = m(input)
```


Sparse layers

```
class torch.nn.Embedding(num_embeddings, embedding_dim, padding_idx=None, max_norm=None, norm_type
```

一个保存了固定字典和大小的简单查找表。

这个模块常用来保存词嵌入和用下标检索它们。模块的输入是一个下标的列表，输出是对应的词嵌入。

参数：

- `num_embeddings(int)` - 嵌入字典的大小
- `embedding_dim(int)` - 每个嵌入向量的大小
- `padding_idx(int, optional)` - 如果提供的话，输出遇到此下标时用零填充
- `max_norm(float, optional)` - 如果提供的话，会重新归一化词嵌入，使它们的范数小于提供的值
- `norm_type(float, optional)` - 对于max_norm选项计算p范数时的p
- `scale_grad_by_freq(boolean, optional)` - 如果提供的话，会根据字典中单词频率缩放  [v: latest](#) ▼

变量：

- **weight** (*Tensor*) -形状为(num_embeddings, embedding_dim)的模块中可学习的权值

形状:

- **输入:** LongTensor (N, W), N = mini-batch, W = 每个mini-batch中提取的下标数
- **输出:** ($N, W, embedding_dim$)

例子:

```
>>> # an Embedding module containing 10 tensors of size 3
>>> embedding = nn.Embedding(10, 3)
>>> # a batch of 2 samples of 4 indices each
>>> input = Variable(torch.LongTensor([[1,2,4,5],[4,3,2,9]]))
>>> embedding(input)

Variable containing:
(0 ,.,.) =
-1.0822  1.2522  0.2434
 0.8393 -0.6062 -0.3348
 0.6597  0.0350  0.0837
 0.5521  0.9447  0.0498

(1 ,.,.) =
 0.6597  0.0350  0.0837
-0.1527  0.0877  0.4260
 0.8393 -0.6062 -0.3348
-0.8738 -0.9054  0.4281
[torch.FloatTensor of size 2x4x3]

>>> # example with padding_idx
>>> embedding = nn.Embedding(10, 3, padding_idx=0)
>>> input = Variable(torch.LongTensor([[0,2,0,5]]))
>>> embedding(input)

Variable containing:
(0 ,.,.) =
 0.0000  0.0000  0.0000
 0.3452  0.4937 -0.9361
 0.0000  0.0000  0.0000
 0.0706 -2.1962 -0.6276
[torch.FloatTensor of size 1x4x3]
```

Distance functions

```
class torch.nn.PairwiseDistance(p=2, eps=1e-06)
```

按批计算向量v1, v2之间的距离:

$$\|x\|_p := \left(\sum_{i=1}^n |x_i|^p \right)^{1/p}$$

参数:

- **x** (*Tensor*): 包含两个输入batch的张量
- **p** (real): 范数次数, 默认值: 2

形状:

- 输入: (N, D) , 其中 D =向量维数
- 输出: $(N, 1)$

```
>>> pdist = nn.PairwiseDistance(2)
>>> input1 = autograd.Variable(torch.randn(100, 128))
>>> input2 = autograd.Variable(torch.randn(100, 128))
>>> output = pdist(input1, input2)
```

Loss functions

基本用法:

```
criterion = LossCriterion() #构造函数有自己的参数
loss = criterion(x, y) #调用标准时也有参数
```

计算出来的结果已经对 `mini-batch` 取了平均。

`class torch.nn.L1Loss(size_average=True)`[\[source\]](#)

创建一个衡量输入 `x` (模型预测输出) 和目标 `y` 之间差的绝对值的平均值的标准。

$$\text{loss}(x, y) = 1/n \sum |x_i - y_i|$$

- `x` 和 `y` 可以是任意形状, 每个包含 `n` 个元素。
- 对 `n` 个元素对应的差值的绝对值求和, 得出来的结果除以 `n`。
- 如果在创建 `L1Loss` 实例的时候在构造函数中传入 `size_average=False`, 那么求出来的绝对值的和将不会除以 `n`

`class torch.nn.MSELoss(size_average=True)`[\[source\]](#)

创建一个衡量输入 `x` (模型预测输出) 和目标 `y` 之间均方误差标准。

$$\text{loss}(x, y) = 1/n \sum (x_i - y_i)^2$$

- `x` 和 `y` 可以是任意形状, 每个包含 `n` 个元素。
- 对 `n` 个元素对应的差值的绝对值求和, 得出来的结果除以 `n`。
- 如果在创建 `MSELoss` 实例的时候在构造函数中传入 `size_average=False`, 那么求出来的平方和将不会除以 `n`

`class torch.nn.CrossEntropyLoss(weight=None, size_average=True)`

[\[source\]](#)

 v: latest ▼

此标准将 `LogSoftMax` 和 `NLLLoss` 集成到一个类中。

当训练一个多类分类器的时候，这个方法是十分有用的。

- `weight(tensor)`: 1-D tensor, `n` 个元素，分别代表 `n` 类的权重，如果你的训练样本很不均衡的话，是非常有用的。默认值为None。

调用时参数：

- `input`: 包含每个类的得分, 2-D tensor, `shape` 为 `batch*n`
- `target`: 大小为 `n` 的 1-D tensor, 包含类别的索引(0到 `n-1`)。

Loss可以表述为以下形式：

$$loss(x, class) = -\log \frac{\exp(x[class])}{\sum_j \exp(x[j])} = -x[class] + \log(\sum_j \exp(x[j]))$$

当 `weight` 参数被指定的时候, `loss` 的计算公式变为：

$$loss(x, class) = weights[class] * (-x[class] + \log(\sum_j \exp(x[j])))$$

计算出的 `loss` 对 `mini-batch` 的大小取了平均。

形状(`shape`):

- Input: (N,C) `C` 是类别的数量
- Target: (N) `N` 是 `mini-batch` 的大小, $0 \leq targets[i] \leq C-1$

`class torch.nn.NLLLoss(weight=None, size_average=True)`[\[source\]](#)

负的 `log likelihood loss` 损失。用于训练一个 `n` 类分类器。

如果提供的话, `weight` 参数应该是一个 1-D tensor, 里面的值对应类别的权重。当你的训练集样本不均衡的话, 使用这个参数是非常有用的。

输入是一个包含类别 `log-probabilities` 的 2-D tensor, 形状是 `(mini-batch, n)`

可以通过在最后一层加 `LogSoftmax` 来获得类别的 `log-probabilities`。

如果您不想增加一个额外层的话, 您可以使用 `CrossEntropyLoss`。

此 `loss` 期望的 `target` 是类别的索引 (0 to N-1, where N = number of classes)

此 `loss` 可以被表示如下：

$$loss(x, class) = -x[class]$$

如果 `weights` 参数被指定的话, `loss` 可以表示如下:

$$\text{loss}(x, \text{class}) = - \text{weights}[\text{class}] * x[\text{class}]$$

参数说明:

- `weight` (Tensor, optional) – 手动指定每个类别的权重。如果给定的话, 必须是长度为 `nclasses`
- `size_average` (bool, optional) – 默认情况下, 会计算 `mini-batch`loss` 的平均值。然而, 如果 `size_average=False` 那么将会把 `mini-batch` 中所有样本的 `loss` 累加起来。

形状:

- Input: (N,C), `C` 是类别的个数
- Target: (N), `target` 中每个值的大小满足 `0 <= targets[i] <= C-1`

例子:

```
m = nn.LogSoftmax()
loss = nn.NLLLoss()
# input is of size nBatch x nClasses = 3 x 5
input = autograd.Variable(torch.randn(3, 5), requires_grad=True)
# each element in target has to have 0 <= value < nclasses
target = autograd.Variable(torch.LongTensor([1, 0, 4]))
output = loss(m(input), target)
output.backward()
```

class torch.nn.NLLLoss2d(weight=None, size_average=True)[\[source\]](#)

对于图片的 `negative log likelihood loss`。计算每个像素的 `NLL loss`。

参数说明:

- `weight` (Tensor, optional) – 用来作为每类的权重, 如果提供的话, 必须为 `1-D` tensor, 大小为 `C`: 类别的个数。
- `size_average` – 默认情况下, 会计算 `mini-batch` `loss` 均值。如果设置为 `False` 的话, 将会累加 `mini-batch` 中所有样本的 `loss` 值。默认值: `True`。

形状:

- Input: (N,C,H,W) `C` 类的数量
- Target: (N,H,W) where each value is `0 <= targets[i] <= C-1`

例子:

```
m = nn.Conv2d(16, 32, (3, 3)).float()
loss = nn.NLLLoss2d()
# input is of size nBatch x nClasses x height x width
input = autograd.Variable(torch.randn(3, 16, 10, 10))
# each element in target has to have 0 <= value < nclasses
target = autograd.Variable(torch.LongTensor(3, 8, 8).random_(0, 4))
output = loss(m(input), target)
output.backward()
```

class torch.nn.KLDivLoss(weight=None, size_average=True)[\[source\]](#)

计算 KL 散度损失。

KL散度常用来描述两个分布的距离，并在输出分布的空间上执行直接回归是有用的。

与 `NLLLoss` 一样，给定的输入应该是 `log-probabilities`。然而。和 `NLLLoss` 不同的是，`input` 不限于 `2-D` tensor，因为此标准是基于 `element` 的。

`target` 应该和 `input` 的形状相同。

此loss可以表示为：

$$\text{loss}(x, \text{target}) = \frac{1}{n} \sum_i (\text{target}_i * (\log(\text{target}_i) - x_i))$$

默认情况下，loss会基于 `element` 求平均。如果 `size_average=False` `loss` 会被累加起来。

class torch.nn.BCELoss(weight=None, size_average=True)[\[source\]](#)

计算 `target` 与 `output` 之间的二进制交叉熵。 $\text{loss}(o, t) = -\frac{1}{n} \sum_i (t[i] \log(o[i]) + (1 - t[i]) \log(1 - o[i]))$ 如果 `weight` 被指定： $\text{loss}(o, t) = -\frac{1}{n} \sum_i \text{weights}[i] (t[i] \log(o[i]) + (1 - t[i]) \log(1 - o[i]))$

这个用于计算 `auto-encoder` 的 `reconstruction error`。注意 $0 \leq \text{target}[i] \leq 1$ 。

默认情况下，loss会基于 `element` 平均，如果 `size_average=False` 的话，`loss` 会被累加。

class torch.nn.MarginRankingLoss(margin=0, size_average=True)[\[source\]](#)

创建一个标准，给定输入 x_1, x_2 两个 1-D mini-batch Tensor's，和一个 y (1-D mini-batch tensor)， y 里面的值只能是 -1 或 1。

如果 `y=1`，代表第一个输入的值应该大于第二个输入的值，如果 `y=-1` 的话，则相反。

`mini-batch` 中每个样本的loss的计算公式如下：

 v: latest ▼

$$\text{loss}(x, y) = \max(0, -y * (x_1 - x_2) + \text{margin})$$

如果 `size_average=True`, 那么求出的 `loss` 将会对 `mini-batch` 求平均, 反之, 求出的 `loss` 会累加。默认情况下, `size_average=True`。

`class torch.nn.HingeEmbeddingLoss(size_average=True)`[\[source\]](#)

给定一个输入 x (2-D mini-batch tensor) 和对应的标签 y (1-D tensor, 1, -1), 此函数用来计算之间的损失值。这个 `loss` 通常用来测量两个输入是否相似, 即: 使用 L1 成对距离。典型是用于学习非线性 `embedding` 或者半监督学习中:

$$loss(x, y) = \frac{1}{n} \sum_i \begin{cases} x_i, & \text{if } y_i == 1 \\ \max(0, margin - x_i), & \text{if } y_i == -1 \end{cases}$$

x 和 y 可以是任意形状, 且都有 `n` 的元素, `loss` 的求和操作作用在所有的元素上, 然后除以 `n`。如果您不想除以 `n` 的话, 可以通过设置 `size_average=False`。

`margin` 的默认值为 1, 可以通过构造函数来设置。

`class torch.nn.MultiLabelMarginLoss(size_average=True)`[\[source\]](#)

计算多标签分类的 `hinge loss` (`margin-based loss`), 计算 `loss` 时需要两个输入: `input x` (`2-D mini-batch Tensor`), 和 `output y` (`2-D tensor` 表示 mini-batch 中样本类别的索引)。

$$loss(x, y) = \frac{1}{x.size(0)} \sum_{i=0, j=0}^{I, J} (\max(0, 1 - (x[y[j]] - x[i])))$$

其中 `I=x.size(0), J=y.size(0)`。对于所有的 `i` 和 `j`, 满足 $y[j] \neq 0, i \neq y[j]$

`x` 和 `y` 必须具有同样的 `size`。

这个标准仅考虑了第一个非零 `y[j] targets` 此标准允许了, 对于每个样本来说, 可以有多个类别。

`class torch.nn.SmoothL1Loss(size_average=True)`[\[source\]](#)

平滑版 `L1 loss`。

`loss` 的公式如下:

$$loss(x, y) = \frac{1}{n} \sum_i \begin{cases} 0.5 * (x_i - y_i)^2, & \text{if } |x_i - y_i| < 1 \\ |x_i - y_i| - 0.5, & \text{otherwise} \end{cases}$$

此 `loss` 对于异常点的敏感性不如 `MSELoss`, 而且, 在某些情况下防止了梯度爆炸, (参照 `Fast R-CNN`)。这个 `loss` 有时也被称为 `Huber loss`。

x和y可以是任何包含 `n` 个元素的tensor。默认情况下，求出来的 `loss` 会除以 `n`，可以通过设置 `size_average=True` 使loss累加。

`class torch.nn.SoftMarginLoss(size_average=True)`[\[source\]](#)

创建一个标准，用来优化2分类的 `logistic loss`。输入为 `x`（一个 2-D mini-batch Tensor）和目标 `y`（一个包含1或-1的Tensor）。

$$loss(x, y) = \frac{1}{x.nelement()} \sum_i (\log(1 + \exp(-y[i] * x[i])))$$

如果求出的 `loss` 不想被平均可以通过设置 `size_average=False`。

`class torch.nn.MultiLabelSoftMarginLoss(weight=None, size_average=True)`[\[source\]](#)

创建一个标准，基于输入x和目标y的 `max-entropy`，优化多标签 `one-versus-all` 的损失。`x`: 2-D mini-batch Tensor; `y`: binary 2D Tensor。对每个mini-batch中的样本，对应的loss为：

$$loss(x, y) = - \frac{1}{x.nelement()} \sum_{i=0}^I y[i] \log \frac{\exp(x[i])}{(1 + \exp(x[i]))} + (1 - y[i]) \log \frac{1}{1 + \exp(x[i])}$$

其中 `I=x.nelement()-1`， $y[i] \in \{0,1\}$ ，`y` 和 `x` 必须要有同样 `size`。

`class torch.nn.CosineEmbeddingLoss(margin=0, size_average=True)`[\[source\]](#)

给定输入 `Tensors`，`x1`，`x2` 和一个标签Tensor `y` (元素的值为1或-1)。此标准使用 `cosine` 距离测量两个输入是否相似，一般用来用来学习非线性 `embedding` 或者半监督学习。

`margin` 应该是-1到1之间的值，建议使用0到0.5。如果没有传入 `margin` 实参，默认值为0。

每个样本的loss是：

$$loss(x, y) = \begin{cases} 1 - \cos(x1, x2), & \text{if } y == 1 \\ \max(0, \cos(x1, x2) - \text{margin}), & \text{if } y == -1 \end{cases}$$

如果 `size_average=True` 求出的loss会对batch求均值，如果 `size_average=False` 的话，则会累加 `loss`。默认情况 `size_average=True`。

`class torch.nn.MultiMarginLoss(p=1, margin=1, weight=None, size_average=True)`[\[source\]](#)

用来计算multi-class classification的hinge loss (margin-based loss)。输入是 `x` (2D mini-batch Tensor)，`y` (1D Tensor)包含类别的索引，`0 <= y <= x.size(1)`。

对每个mini-batch样本：

$$loss(x, y) = \frac{1}{x.size(0)} \sum_{i=0}^I (max(0, margin - x[y] + x[i])^p)$$

其中 `I=x.size(0)` $i \neq y$ 。 可选的，如果您不想所有的类拥有同样的权重的话，您可以通过在构造函数中传入 `weights` 参数来解决这个问题， `weights` 是一个1D权重Tensor。

传入weights后， loss函数变为：

$$loss(x, y) = \frac{1}{x.size(0)} \sum_i max(0, w[y] * (margin - x[y] - x[i]))^p$$

默认情况下，求出的loss会对mini-batch取平均，可以通过设置 `size_average=False` 来取消取平均操作。

Vision layers

`class torch.nn.PixelShuffle(upscale_factor)`[\[source\]](#)

将shape为 $[N, Cr^2, H, W]$ 的 `Tensor` 重新排列为shape为 $[N, C, Hr, W*r]$ 的Tensor。 当使用 `stride=1/r` 的sub-pixel卷积的时候，这个方法是非常有用的。

请看[paperReal-Time Single Image and Video Super-Resolution Using an Efficient Sub-Pixel Convolutional Neural Network by Shi et. al \(2016\)](#) 获取详细信息。

参数说明：

- `upscale_factor` (int) – 增加空间分辨率的因子

Shape:

- Input: $[N, C * upscale_factor^2, H, W]$
- Output: $[N, C, H * upscale_factor, W * upscale_factor]$

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])
```

`class torch.nn.UpsamplingNearest2d(size=None, scale_factor=None)`

[\[source\]](#)

 v: latest ▼

对于多channel 输入 进行 `2-D` 最近邻上采样。

可以通过 `size` 或者 `scale_factor` 来指定上采样后的图片大小。

当给定 `size` 时, `size` 的值将会是输出图片的大小。

参数:

- `size` (tuple, optional) – 一个包含两个整数的元组 (`H_out`, `W_out`)指定了输出的长宽
- `scale_factor` (int, optional) – 长和宽的一个乘子

形状:

- Input: (`N`,`C`,`H_in`,`W_in`)
- Output: (`N`,`C`,`H_out`,`W_out`) $H_{out} = \text{floor}(H_{in} * \text{scale_factor})$ $W_{out} = \text{floor}(W_{in} * \text{scale_factor})$

例子:

```
>>> inp
Variable containing:
(0 ,0 ,...,) =
  1  2
  3  4
[torch.FloatTensor of size 1x1x2x2]

>>> m = nn.UpsamplingNearest2d(scale_factor=2)
>>> m(inp)
Variable containing:
(0 ,0 ,...,) =
  1  1  2  2
  1  1  2  2
  3  3  4  4
  3  3  4  4
[torch.FloatTensor of size 1x1x4x4]
```

`class torch.nn.UpsamplingBilinear2d(size=None, scale_factor=None)`

[\[source\]](#)

对于多channel 输入 进行 `2-D` `bilinear` 上采样。

可以通过 `size` 或者 `scale_factor` 来指定上采样后的图片大小。

当给定 `size` 时, `size` 的值将会是输出图片的大小。

参数:

- `size` (tuple, optional) – 一个包含两个整数的元组 (`H_out`, `W_out`)指定了输出的长宽
- `scale_factor` (int, optional) – 长和宽的一个乘子

 [v: latest](#) ▼

形状:

- Input: (N,C,H_in,W_in)
- Output: (N,C,H_out,W_out) $H_{out} = \text{floor}(H_{in} * \text{scale_factor})$ $W_{out} = \text{floor}(W_{in} * \text{scale_factor})$

例子:

```
>>> inp
Variable containing:
(0 ,0 ,.,.) =
  1  2
  3  4
[torch.FloatTensor of size 1x1x2x2]

>>> m = nn.UpsamplingBilinear2d(scale_factor=2)
>>> m(inp)
Variable containing:
(0 ,0 ,.,.) =
  1.0000  1.3333  1.6667  2.0000
  1.6667  2.0000  2.3333  2.6667
  2.3333  2.6667  3.0000  3.3333
  3.0000  3.3333  3.6667  4.0000
[torch.FloatTensor of size 1x1x4x4]
```

Multi-GPU layers

class torch.nn.DataParallel(module, device_ids=None, output_device=None, dim=0)[\[source\]](#)

在模块级别上实现数据并行。

此容器通过将 `mini-batch` 划分到不同的设备上来实现给定 `module` 的并行。在 `forward` 过程中, `module` 会在每个设备上复制一遍, 每个副本都会处理部分输入。在 `backward` 过程中, 副本上的梯度会累加到原始 `module` 上。

batch的大小应该大于所使用的GPU的数量。还应当是GPU个数的整数倍, 这样划分出来的每一块都会有相同的样本数量。

请看: [Use nn.DataParallel instead of multiprocessing](#)

除了 `Tensor`, 任何位置参数和关键字参数都可以传到DataParallel中。所有的变量会通过指定的 `dim` 来划分 (默认值为0)。原始类型将会被广播, 但是所有的其它类型都会被浅复制。所以如果在模型的 `forward` 过程中写入的话, 将会被损坏。

参数说明:

- module – 要被并行的module
- device_ids – CUDA设备, 默认为所有设备。
- output_device – 输出设备 (默认为device_ids[0])

例子:

```
net = torch.nn.DataParallel(model, device_ids=[0, 1, 2])
output = net(input_var)
```

Utilities

工具函数

torch.nn.utils.clip_grad_norm(parameters, max_norm, norm_type=2)[\[source\]](#)

Clips gradient norm of an iterable of parameters.

正则项的值由所有的梯度计算出来，就像他们连成一个向量一样。梯度被 `in-place operation` 修改。

参数说明: - parameters (Iterable[Variable]) – 可迭代的 `Variables`，它们的梯度即将被标准化。 - max_norm (float or int) – `clip` 后，`gradients` p-norm 值 - norm_type (float or int) – 标准化的类型，p-norm. 可以是 `inf` 代表 infinity norm.

关于norm

返回值:

所有参数的p-norm值。

torch.nn.utils.rnn.PackedSequence(_cls, data, batch_sizes)[\[source\]](#)

Holds the data and list of batch_sizes of a packed sequence.

All RNN modules accept packed sequences as inputs. 所有的 `RNN` 模块都接收这种被包裹后的序列作为它们的输入。

NOTE: 这个类的实例不能手动创建。它们只能被 `pack_padded_sequence()` 实例化。

参数说明:

- data (Variable) – 包含打包后序列的 `Variable`。
- batch_sizes (list[int]) – 包含 `mini-batch` 中每个序列长度的列表。

torch.nn.utils.rnn.pack_padded_sequence(input, lengths, batch_first=False)[\[source\]](#)

这里的 `pack`，理解成压紧比较好。将一个填充过的变长序列压紧。（填充时候，会有冗余，所以压紧一下）

 v: latest ▼

输入的形状可以是($T \times B \times *$)。 T 是最长序列长度, B 是 `batch size`, $*$ 代表任意维度(可以是0)。如果 `batch_first=True` 的话, 那么相应的 `input size` 就是 ($B \times T \times *$)。

`Variable` 中保存的序列, 应该按序列长度的长短排序, 长的在前, 短的后。即 `input[:,0]` 代表的是最长的序列, `input[:, B-1]` 保存的是最短的序列。

NOTE: 只要是维度大于等于2的 `input` 都可以作为这个函数的参数。你可以用它来打包 `labels`, 然后用 `RNN` 的输出和打包后的 `labels` 来计算 `loss`。通过 `PackedSequence` 对象的 `.data` 属性可以获取 `Variable`。

参数说明:

- `input (Variable)` – 变长序列 被填充后的 batch
- `lengths (list[int])` – `Variable` 中每个序列的长度。
- `batch_first (bool, optional)` – 如果是 `True`, `input` 的形状应该是 $B \times T \times \text{size}$ 。

返回值:

一个 `PackedSequence` 对象。

`torch.nn.utils.rnn.pad_packed_sequence(sequence, batch_first=False)`[\[source\]](#)

填充 `packed_sequence`。

上面提到的函数的功能是将一个填充后的变长序列压紧。这个操作和 `pack_padded_sequence()` 是相反的。把压紧的序列再填充回来。

返回的 `Variable` 的值的 `size` 是 $T \times B \times *$, T 是最长序列的长度, B 是 `batch_size`, 如果 `batch_first=True`, 那么返回值是 $B \times T \times *$ 。

Batch 中的元素将会以它们长度的逆序排列。

参数说明:

- `sequence (PackedSequence)` – 将要被填充的 batch
- `batch_first (bool, optional)` – 如果为 `True`, 返回的数据的格式为 $B \times T \times *$ 。

返回值: 一个 tuple, 包含被填充后的序列, 和 batch 中序列的长度列表。

例子:

```

import torch
import torch.nn as nn
from torch.autograd import Variable
from torch.nn import utils as nn_utils

batch_size = 2
max_length = 3
hidden_size = 2
n_layers = 1

tensor_in = torch.FloatTensor([[1, 2, 3], [1, 0, 0]]).resize_(2,3,1)
tensor_in = Variable( tensor_in ) #[batch, seq, feature], [2, 3, 1]
seq_lengths = [3,1] # List of integers holding information about the batch size at each sequence

# pack it
pack = nn_utils.rnn.pack_padded_sequence(tensor_in, seq_lengths, batch_first=True)

# initialize
rnn = nn.RNN(1, hidden_size, n_layers, batch_first=True)
h0 = Variable(torch.randn(n_layers, batch_size, hidden_size))

#forward
out, _ = rnn(pack, h0)

# unpack
unpacked = nn_utils.rnn.pad_packed_sequence(out)
print(unpacked)

```

关于packed_sequence