pytorch / **pytorch**

Branch: master ▾ **pytorch** / torch / nn / modules / **conv.py** / Jump to ▾ Find file Copy path

driazati Remove weak script (#22212) 10c4b98 12 days ago

40 contributors and others

934 lines (758 sloc) 45.1 KB Raw Blame History

Jump to definition is available! Beta

Navigate your code with ease. In select public repositories, you can now click on function and method calls to jump to their definitions in the same repository. Learn more

● You're using jump to definition to discover and navigate code. Opt out Beta Learn more or give us feedback

```python
# coding=utf-8
import math
import torch
from torch.nn.parameter import Parameter
from .. import functional as F
from .. import init
from .module import Module
from .utils import _single, _pair, _triple
from ..._jit_internal import List


class _ConvNd(Module):

    __constants__ = ['stride', 'padding', 'dilation', 'groups', 'bias',
                     'padding_mode', 'output_padding', 'in_channels',
                     'out_channels', 'kernel_size']

    def __init__(self, in_channels, out_channels, kernel_size, stride,
                 padding, dilation, transposed, output_padding,
                 groups, bias, padding_mode):
        super(_ConvNd, self).__init__()
        if in_channels % groups != 0:
            raise ValueError('in_channels must be divisible by groups')
        if out_channels % groups != 0:
            raise ValueError('out_channels must be divisible by groups')
        self.in_channels = in_channels
        self.out_channels = out_channels
        self.kernel_size = kernel_size
        self.stride = stride
        self.padding = padding
        self.dilation = dilation
        self.transposed = transposed
        self.output_padding = output_padding
        self.groups = groups
        self.padding_mode = padding_mode
        if transposed:
            self.weight = Parameter(torch.Tensor(
                in_channels, out_channels // groups, *kernel_size))
        else:
            self.weight = Parameter(torch.Tensor(
                out_channels, in_channels // groups, *kernel_size))
        if bias:
            self.bias = Parameter(torch.Tensor(out_channels))
        else:
            self.register_parameter('bias', None)
        self.reset_parameters()

    def reset_parameters(self):
        init.kaiming_uniform_(self.weight, a=math.sqrt(5))
        if self.bias is not None:
            fan_in, _ = init._calculate_fan_in_and_fan_out(self.weight)
            bound = 1 / math.sqrt(fan_in)
            init.uniform_(self.bias, -bound, bound)

    def extra_repr(self):
        s = ('{in_channels}, {out_channels}, kernel_size={kernel_size}'
             ', stride={stride}')
        if self.padding != (0,) * len(self.padding):
            s += ', padding={padding}'
        if self.dilation != (1,) * len(self.dilation):
            s += ', dilation={dilation}'
        if self.output_padding != (0,) * len(self.output_padding):
            s += ', output_padding={output_padding}'
        if self.groups != 1:
            s += ', groups={groups}'
        if self.bias is None:
            s += ', bias=False'
        return s.format(**self.__dict__)

    def __setstate__(self, state):
        super(_ConvNd, self).__setstate__(state)
        if not hasattr(self, 'padding_mode'):
            self.padding_mode = 'zeros'


class Conv1d(_ConvNd):
    r"""Applies a 1D convolution over an input signal composed of several input
    planes.

    In the simplest case, the output value of the layer with input size
    :math:`(N, C_{\text{in}}, L)` and output :math:`(N, C_{\text{out}}, L_{\text{out}})` can be
    precisely described as:

    .. math::
        \text{out}(N_i, C_{\text{out}_j}) = \text{bias}(C_{\text{out}_j}) +
        \sum_{k = 0}^{C_{in} - 1} \text{weight}(C_{\text{out}_j}, k)
        \star \text{input}(N_i, k)

    where :math:`\star` is the valid `cross-correlation`_ operator,
    :math:`N` is a batch size, :math:`C` denotes a number of channels,
    :math:`L` is a length of signal sequence.

    * :attr:`stride` controls the stride for the cross-correlation, a single
      number or a one-element tuple.

    * :attr:`padding` controls the amount of implicit zero-paddings on both sides
      for :attr:`padding` number of points.

    * :attr:`dilation` controls the spacing between the kernel points; also
      known as the à trous algorithm. It is harder to describe, but this `link`_
      has a nice visualization of what :attr:`dilation` does.

    * :attr:`groups` controls the connections between inputs and outputs.
      :attr:`in_channels` and :attr:`out_channels` must both be divisible by
      :attr:`groups`. For example,

        * At groups=1, all inputs are convolved to all outputs.
        * At groups=2, the operation becomes equivalent to having two conv
          layers side by side, each seeing half the input channels,
          and producing half the output channels, and both subsequently
          concatenated.
        * At groups= :attr:`in_channels`, each input channel is convolved with
          its own set of filters,
          of size
          :math:`\left\lfloor\frac{out\_channels}{in\_channels}\right\rfloor`.

    .. note::

        Depending of the size of your kernel, several (of the last)
        columns of the input might be lost, because it is a valid
        `cross-correlation`_, and not a full `cross-correlation`_.
        It is up to the user to add proper padding.

    .. note::

        When `groups == in_channels` and `out_channels == K * in_channels`,
        where `K` is a positive integer, this operation is also termed in
        literature as depthwise convolution.

        In other words, for an input of size :math:`(N, C_{in}, L_{in})`,
        a depthwise convolution with a depthwise multiplier `K`, can be constructed by arguments
        :math:`(C_\text{in}=C_{in}, C_\text{out}=C_{in} \times K, ..., \text{groups}=C_{in})`.

    .. include:: cudnn_deterministic.rst

    Args:
        in_channels (int): Number of channels in the input image
        out_channels (int): Number of channels produced by the convolution
        kernel_size (int or tuple): Size of the convolving kernel
        stride (int or tuple, optional): Stride of the convolution. Default: 1
        padding (int or tuple, optional): Zero-padding added to both sides of
            the input. Default: 0
        padding_mode (string, optional). Accepted values `zeros` and `circular` Default: `zeros`
        dilation (int or tuple, optional): Spacing between kernel
            elements. Default: 1
        groups (int, optional): Number of blocked connections from input
            channels to output channels. Default: 1
        bias (bool, optional): If ``True``, adds a learnable bias to the output. Default: ``True``

    Shape:
        - Input: :math:`(N, C_{in}, L_{in})`
        - Output: :math:`(N, C_{out}, L_{out})` where

          .. math::
              L_{out} = \left\lfloor\frac{L_{in} + 2 \times \text{padding} - \text{dilation}
                        \times (\text{kernel\_size} - 1) - 1}{\text{stride}} + 1\right\rfloor

    Attributes:
        weight (Tensor): the learnable weights of the module of shape
            :math:`(\text{out\_channels}, \frac{\text{in\_channels}}{\text{groups}}, \text{kernel\_size})`.
            The values of these weights are sampled from
            :math:`\mathcal{U}(-\sqrt{k}, \sqrt{k})` where
            :math:`k = \frac{1}{C_\text{in} * \text{kernel\_size}}`
        bias (Tensor):   the learnable bias of the module of shape
            (out_channels). If :attr:`bias` is ``True``, then the values of these weights are
            sampled from :math:`\mathcal{U}(-\sqrt{k}, \sqrt{k})` where
            :math:`k = \frac{1}{C_\text{in} * \text{kernel\_size}}`

    Examples::

        >>> m = nn.Conv1d(16, 33, 3, stride=2)
        >>> input = torch.randn(20, 16, 50)
        >>> output = m(input)

    .. _cross-correlation:
        https://en.wikipedia.org/wiki/Cross-correlation

    .. _link:
        https://github.com/vdumoulin/conv_arithmetic/blob/master/README.md
    """

    def __init__(self, in_channels, out_channels, kernel_size, stride=1,
                 padding=0, dilation=1, groups=1,
                 bias=True, padding_mode='zeros'):
```

```
        kernel_size = _single(kernel_size)
        stride = _single(stride)
        padding = _single(padding)
        dilation = _single(dilation)
        super(Conv1d, self).__init__(
            in_channels, out_channels, kernel_size, stride, padding, dilation,
            False, _single(0), groups, bias, padding_mode)

    def forward(self, input):
        if self.padding_mode == 'circular':
            expanded_padding = ((self.padding[0] + 1) // 2, self.padding[0] // 2)
            return F.conv1d(F.pad(input, expanded_padding, mode='circular'),
                            self.weight, self.bias, self.stride,
                            _single(0), self.dilation, self.groups)
        return F.conv1d(input, self.weight, self.bias, self.stride,
                        self.padding, self.dilation, self.groups)


class Conv2d(_ConvNd):
    r"""Applies a 2D convolution over an input signal composed of several input
    planes.

    In the simplest case, the output value of the layer with input size
    :math:`(N, C_{\text{in}}, H, W)` and output :math:`(N, C_{\text{out}}, H_{\text{out}}, W_{\text{out}})`
    can be precisely described as:

    .. math::
        \text{out}(N_i, C_{\text{out}_j}) = \text{bias}(C_{\text{out}_j}) +
        \sum_{k = 0}^{C_{\text{in}} - 1} \text{weight}(C_{\text{out}_j}, k) \star \text{input}(N_i, k)


    where :math:`\star` is the valid 2D `cross-correlation`_ operator,
    :math:`N` is a batch size, :math:`C` denotes a number of channels,
    :math:`H` is a height of input planes in pixels, and :math:`W` is
    width in pixels.

    * :attr:`stride` controls the stride for the cross-correlation, a single
      number or a tuple.

    * :attr:`padding` controls the amount of implicit zero-paddings on both
      sides for :attr:`padding` number of points for each dimension.

    * :attr:`dilation` controls the spacing between the kernel points; also
      known as the à trous algorithm. It is harder to describe, but this `link`_
      has a nice visualization of what :attr:`dilation` does.

    * :attr:`groups` controls the connections between inputs and outputs.
      :attr:`in_channels` and :attr:`out_channels` must both be divisible by
      :attr:`groups`. For example,

        * At groups=1, all inputs are convolved to all outputs.
        * At groups=2, the operation becomes equivalent to having two conv
```

```
          layers side by side, each seeing half the input channels,
          and producing half the output channels, and both subsequently
          concatenated.
        * At groups= :attr:`in_channels`, each input channel is convolved with
          its own set of filters, of size:
          :math:`\left\lfloor\frac{out\_channels}{in\_channels}\right\rfloor`.

    The parameters :attr:`kernel_size`, :attr:`stride`, :attr:`padding`, :attr:`dilation` can either be:

        - a single ``int`` -- in which case the same value is used for the height and width dimension
        - a ``tuple`` of two ints -- in which case, the first `int` is used for the height dimension,
          and the second `int` for the width dimension

    .. note::

         Depending of the size of your kernel, several (of the last)
         columns of the input might be lost, because it is a valid `cross-correlation`_,
         and not a full `cross-correlation`_.
         It is up to the user to add proper padding.

    .. note::

        When `groups == in_channels` and `out_channels == K * in_channels`,
        where `K` is a positive integer, this operation is also termed in
        literature as depthwise convolution.

        In other words, for an input of size :math:`(N, C_{in}, H_{in}, W_{in})`,
        a depthwise convolution with a depthwise multiplier `K`, can be constructed by arguments
        :math:`(in\_channels=C_{in}, out\_channels=C_{in} \times K, ..., groups=C_{in})`.

    .. include:: cudnn_deterministic.rst

    Args:
        in_channels (int): Number of channels in the input image
        out_channels (int): Number of channels produced by the convolution
        kernel_size (int or tuple): Size of the convolving kernel
        stride (int or tuple, optional): Stride of the convolution. Default: 1
        padding (int or tuple, optional): Zero-padding added to both sides of the input. Default: 0
        padding_mode (string, optional). Accepted values `zeros` and `circular` Default: `zeros`
        dilation (int or tuple, optional): Spacing between kernel elements. Default: 1
        groups (int, optional): Number of blocked connections from input channels to output channels. Default: 1
        bias (bool, optional): If ``True``, adds a learnable bias to the output. Default: ``True``

    Shape:
        - Input: :math:`(N, C_{in}, H_{in}, W_{in})`
        - Output: :math:`(N, C_{out}, H_{out}, W_{out})` where

          .. math::
              H_{out} = \left\lfloor\frac{H_{in}  + 2 \times \text{padding}[0] - \text{dilation}[0]
                        \times (\text{kernel\_size}[0] - 1) - 1}{\text{stride}[0]} + 1\right\rfloor

          .. math::
              W_{out} = \left\lfloor\frac{W_{in}  + 2 \times \text{padding}[1] - \text{dilation}[1]
                        \times (\text{kernel\_size}[1] - 1) - 1}{\text{stride}[1]} + 1\right\rfloor

    Attributes:
        weight (Tensor): the learnable weights of the module of shape
                         :math:`(\text{out\_channels}, \frac{\text{in\_channels}}{\text{groups}},`
                         :math:`\text{kernel\_size[0]}, \text{kernel\_size[1]})`.
                         The values of these weights are sampled from
                         :math:`\mathcal{U}(-\sqrt{k}, \sqrt{k})` where
                         :math:`k = \frac{1}{C_\text{in} * \prod_{i=0}^{1}\text{kernel\_size}[i]}`
        bias (Tensor):   the learnable bias of the module of shape (out_channels). If :attr:`bias` is ``True``,
                         then the values of these weights are
                         sampled from :math:`\mathcal{U}(-\sqrt{k}, \sqrt{k})` where
                         :math:`k = \frac{1}{C_\text{in} * \prod_{i=0}^{1}\text{kernel\_size}[i]}`

    Examples::

        >>> # With square kernels and equal stride
        >>> m = nn.Conv2d(16, 33, 3, stride=2)
        >>> # non-square kernels and unequal stride and with padding
        >>> m = nn.Conv2d(16, 33, (3, 5), stride=(2, 1), padding=(4, 2))
        >>> # non-square kernels and unequal stride and with padding and dilation
        >>> m = nn.Conv2d(16, 33, (3, 5), stride=(2, 1), padding=(4, 2), dilation=(3, 1))
        >>> input = torch.randn(20, 16, 50, 100)
        >>> output = m(input)

    .. _cross-correlation:
        https://en.wikipedia.org/wiki/Cross-correlation

    .. _link:
        https://github.com/vdumoulin/conv_arithmetic/blob/master/README.md
    """
    def __init__(self, in_channels, out_channels, kernel_size, stride=1,
                 padding=0, dilation=1, groups=1,
                 bias=True, padding_mode='zeros'):
        kernel_size = _pair(kernel_size)
        stride = _pair(stride)
        padding = _pair(padding)
        dilation = _pair(dilation)
        super(Conv2d, self).__init__(
            in_channels, out_channels, kernel_size, stride, padding, dilation,
            False, _pair(0), groups, bias, padding_mode)

    def forward(self, input):
        if self.padding_mode == 'circular':
            expanded_padding = ((self.padding[1] + 1) // 2, self.padding[1] // 2,
                                (self.padding[0] + 1) // 2, self.padding[0] // 2)
            return F.conv2d(F.pad(input, expanded_padding, mode='circular'),
                            self.weight, self.bias, self.stride,
                            _pair(0), self.dilation, self.groups)
        return F.conv2d(input, self.weight, self.bias, self.stride,
                        self.padding, self.dilation, self.groups)
```

```


class Conv3d(_ConvNd):
    r"""Applies a 3D convolution over an input signal composed of several input
    planes.

    In the simplest case, the output value of the layer with input size :math:`(N, C_{in}, D, H, W)`
    and output :math:`(N, C_{out}, D_{out}, H_{out}, W_{out})` can be precisely described as:

    .. math::
        out(N_i, C_{out_j}) = bias(C_{out_j}) +
                                \sum_{k = 0}^{C_{in} - 1} weight(C_{out_j}, k) \star input(N_i, k)

    where :math:`\star` is the valid 3D `cross-correlation`_ operator

    * :attr:`stride` controls the stride for the cross-correlation.

    * :attr:`padding` controls the amount of implicit zero-paddings on both
      sides for :attr:`padding` number of points for each dimension.

    * :attr:`dilation` controls the spacing between the kernel points; also known as the à trous algorithm.
      It is harder to describe, but this `link`_ has a nice visualization of what :attr:`dilation` does.

    * :attr:`groups` controls the connections between inputs and outputs.
      :attr:`in_channels` and :attr:`out_channels` must both be divisible by
      :attr:`groups`. For example,

        * At groups=1, all inputs are convolved to all outputs.
        * At groups=2, the operation becomes equivalent to having two conv
          layers side by side, each seeing half the input channels,
          and producing half the output channels, and both subsequently
          concatenated.
        * At groups= :attr:`in_channels`, each input channel is convolved with
          its own set of filters, of size
          :math:`\left\lfloor\frac{out\_channels}{in\_channels}\right\rfloor`.

    The parameters :attr:`kernel_size`, :attr:`stride`, :attr:`padding`, :attr:`dilation` can either be:

        - a single ``int`` -- in which case the same value is used for the depth, height and width dimension
        - a ``tuple`` of three ints -- in which case, the first `int` is used for the depth dimension,
          the second `int` for the height dimension and the third `int` for the width dimension

    .. note::

         Depending of the size of your kernel, several (of the last)
         columns of the input might be lost, because it is a valid `cross-correlation`_,
         and not a full `cross-correlation`_.
         It is up to the user to add proper padding.

    .. note::

        When `groups == in_channels` and `out_channels == K * in_channels`,
        where `K` is a positive integer, this operation is also termed in
        literature as depthwise convolution.

        In other words, for an input of size :math:`(N, C_{in}, D_{in}, H_{in}, W_{in})`,
        a depthwise convolution with a depthwise multiplier `K`, can be constructed by arguments
        :math:`(in\_channels=C_{in}, out\_channels=C_{in} \times K, ..., groups=C_{in})`.

    .. include:: cudnn_deterministic.rst

    Args:
        in_channels (int): Number of channels in the input image
        out_channels (int): Number of channels produced by the convolution
        kernel_size (int or tuple): Size of the convolving kernel
        stride (int or tuple, optional): Stride of the convolution. Default: 1
        padding (int or tuple, optional): Zero-padding added to all three sides of the input. Default: 0
        padding_mode (string, optional). Accepted values `zeros` and `circular` Default: `zeros`
        dilation (int or tuple, optional): Spacing between kernel elements. Default: 1
        groups (int, optional): Number of blocked connections from input channels to output channels. Default: 1
        bias (bool, optional): If ``True``, adds a learnable bias to the output. Default: ``True``

    Shape:
        - Input: :math:`(N, C_{in}, D_{in}, H_{in}, W_{in})`
        - Output: :math:`(N, C_{out}, D_{out}, H_{out}, W_{out})` where

          .. math::
              D_{out} = \left\lfloor\frac{D_{in} + 2 \times \text{padding}[0] - \text{dilation}[0]
                    \times (\text{kernel\_size}[0] - 1) - 1}{\text{stride}[0]} + 1\right\rfloor

          .. math::
              H_{out} = \left\lfloor\frac{H_{in} + 2 \times \text{padding}[1] - \text{dilation}[1]
                    \times (\text{kernel\_size}[1] - 1) - 1}{\text{stride}[1]} + 1\right\rfloor

          .. math::
              W_{out} = \left\lfloor\frac{W_{in} + 2 \times \text{padding}[2] - \text{dilation}[2]
                    \times (\text{kernel\_size}[2] - 1) - 1}{\text{stride}[2]} + 1\right\rfloor

    Attributes:
        weight (Tensor): the learnable weights of the module of shape
                         :math:`(\text{out\_channels}, \frac{\text{in\_channels}}{\text{groups}},`
                         :math:`\text{kernel\_size[0]}, \text{kernel\_size[1]}, \text{kernel\_size[2]})`.
                         The values of these weights are sampled from
                         :math:`\mathcal{U}(-\sqrt{k}, \sqrt{k})` where
                         :math:`k = \frac{1}{C_\text{in} * \prod_{i=0}^{2}\text{kernel\_size}[i]}`
        bias (Tensor):   the learnable bias of the module of shape (out_channels). If :attr:`bias` is ``True``,
                         then the values of these weights are
                         sampled from :math:`\mathcal{U}(-\sqrt{k}, \sqrt{k})` where
                         :math:`k = \frac{1}{C_\text{in} * \prod_{i=0}^{2}\text{kernel\_size}[i]}`

    Examples::

        >>> # With square kernels and equal stride
        >>> m = nn.Conv3d(16, 33, 3, stride=2)
        >>> # non-square kernels and unequal stride and with padding
        >>> m = nn.Conv3d(16, 33, (3, 5, 2), stride=(2, 1, 1), padding=(4, 2, 0))
        >>> input = torch.randn(20, 16, 10, 50, 100)
        >>> output = m(input)

    .. _cross-correlation:
        https://en.wikipedia.org/wiki/Cross-correlation

    .. _link:
        https://github.com/vdumoulin/conv_arithmetic/blob/master/README.md
    """
    def __init__(self, in_channels, out_channels, kernel_size, stride=1,
                 padding=0, dilation=1, groups=1,
                 bias=True, padding_mode='zeros'):
        kernel_size = _triple(kernel_size)
        stride = _triple(stride)
        padding = _triple(padding)
        dilation = _triple(dilation)
        super(Conv3d, self).__init__(
            in_channels, out_channels, kernel_size, stride, padding, dilation,
            False, _triple(0), groups, bias, padding_mode)

    def forward(self, input):
        if self.padding_mode == 'circular':
            expanded_padding = ((self.padding[2] + 1) // 2, self.padding[2] // 2,
                                (self.padding[1] + 1) // 2, self.padding[1] // 2,
                                (self.padding[0] + 1) // 2, self.padding[0] // 2)
            return F.conv3d(F.pad(input, expanded_padding, mode='circular'),
                            self.weight, self.bias, self.stride, _triple(0),
                            self.dilation, self.groups)
        return F.conv3d(input, self.weight, self.bias, self.stride,
                        self.padding, self.dilation, self.groups)


class _ConvTransposeMixin(object):
    def forward(self, input, output_size=None):
        # type(Tensor, Optional[List[int]]) -> Tensor
        output_padding = self._output_padding(input, output_size, self.stride, self.padding, self.kernel_size)
        func = self._backend.ConvNd(
            self.stride, self.padding, self.dilation, self.transposed,
            output_padding, self.groups)
        if self.bias is None:
            return func(input, self.weight)
        else:
            return func(input, self.weight, self.bias)

    def _output_padding(self, input, output_size, stride, padding, kernel_size):
        # type: (Tensor, Optional[List[int]], List[int], List[int], List[int]) -> List[int]
        if output_size is None:
            ret = _single(self.output_padding)  # converting to list if was not already
        else:
            k = input.dim() - 2
            if len(output_size) == k + 2:
                output_size = output_size[2:]
            if len(output_size) != k:
                raise ValueError(
                    "output_size must have {} or {} elements (got {})"
                    .format(k, k + 2, len(output_size)))

            min_sizes = torch.jit.annotate(List[int], [])
            max_sizes = torch.jit.annotate(List[int], [])
            for d in range(k):
                dim_size = ((input.size(d + 2) - 1) * stride[d] -
                            2 * padding[d] + kernel_size[d])
                min_sizes.append(dim_size)
                max_sizes.append(min_sizes[d] + stride[d] - 1)

            for i in range(len(output_size)):
                size = output_size[i]
                min_size = min_sizes[i]
                max_size = max_sizes[i]
                if size < min_size or size > max_size:
                    raise ValueError((
                        "requested an output size of {}, but valid sizes range "
                        "from {} to {} (for an input of {})").format(
                            output_size, min_sizes, max_sizes, input.size()[2:]))

            res = torch.jit.annotate(List[int], [])
            for d in range(k):
                res.append(output_size[d] - min_sizes[d])

            ret = res
        return ret


class ConvTranspose1d(_ConvTransposeMixin, _ConvNd):
    r"""Applies a 1D transposed convolution operator over an input image
    composed of several input planes.

    This module can be seen as the gradient of Conv1d with respect to its input.
    It is also known as a fractionally-strided convolution or
    a deconvolution (although it is not an actual deconvolution operation).

    * :attr:`stride` controls the stride for the cross-correlation.

    * :attr:`padding` controls the amount of implicit zero-paddings on both
      sides for ``dilation * (kernel_size - 1) - padding`` number of points. See note
      below for details.

    * :attr:`output_padding` controls the additional size added to one side
      of the output shape. See note below for details.

    * :attr:`dilation` controls the spacing between the kernel points; also known as the à trous algorithm.
      It is harder to describe, but this `link`_ has a nice visualization of what :attr:`dilation` does.

    * :attr:`groups` controls the connections between inputs and outputs.
      :attr:`in_channels` and :attr:`out_channels` must both be divisible by
      :attr:`groups`. For example,

        * At groups=1, all inputs are convolved to all outputs.
        * At groups=2, the operation becomes equivalent to having two conv
          layers side by side, each seeing half the input channels,
          and producing half the output channels, and both subsequently
          concatenated.
        * At groups= :attr:`in_channels`, each input channel is convolved with
          its own set of filters (of size
          :math:`\left\lfloor\frac{out\_channels}{in\_channels}\right\rfloor`).

    .. note::

         Depending of the size of your kernel, several (of the last)
         columns of the input might be lost, because it is a valid `cross-correlation`_,
         and not a full `cross-correlation`_.
         It is up to the user to add proper padding.

    .. note::
        The :attr:`padding` argument effectively adds ``dilation * (kernel_size - 1) - padding``
        amount of zero padding to both sizes of the input. This is set so that
        when a :class:`~torch.nn.Conv1d` and a :class:`~torch.nn.ConvTranspose1d`
        are initialized with same parameters, they are inverses of each other in
        regard to the input and output shapes. However, when ``stride > 1``,
        :class:`~torch.nn.Conv1d` maps multiple input shapes to the same output
        shape. :attr:`output_padding` is provided to resolve this ambiguity by
        effectively increasing the calculated output shape on one side. Note
        that :attr:`output_padding` is only used to find output shape, but does
        not actually add zero-padding to output.

    .. include:: cudnn_deterministic.rst

    Args:
        in_channels (int): Number of channels in the input image
        out_channels (int): Number of channels produced by the convolution
        kernel_size (int or tuple): Size of the convolving kernel
        stride (int or tuple, optional): Stride of the convolution. Default: 1
        padding (int or tuple, optional): ``dilation * (kernel_size - 1) - padding`` zero-padding
            will be added to both sides of the input. Default: 0
        output_padding (int or tuple, optional): Additional size added to one side
            of the output shape. Default: 0
        groups (int, optional): Number of blocked connections from input channels to output channels. Default: 1
        bias (bool, optional): If ``True``, adds a learnable bias to the output. Default: ``True``
        dilation (int or tuple, optional): Spacing between kernel elements. Default: 1

    Shape:
        - Input: :math:`(N, C_{in}, L_{in})`
        - Output: :math:`(N, C_{out}, L_{out})` where

```

```
          .. math::
              L_{out} = (L_{in} - 1) \times \text{stride} - 2 \times \text{padding} + \text{dilation}
                        \times (\text{kernel\_size} - 1) + \text{output\_padding} + 1

    Attributes:
        weight (Tensor): the learnable weights of the module of shape
                         :math:`(\text{in\_channels}, \frac{\text{out\_channels}}{\text{groups}},`
                         :math:`\text{kernel\_size})`.
                         The values of these weights are sampled from
                         :math:`\mathcal{U}(-\sqrt{k}, \sqrt{k})` where
                         :math:`k = \frac{1}{C_\text{in} * \text{kernel\_size}}`
        bias (Tensor):   the learnable bias of the module of shape (out_channels).
                         If :attr:`bias` is ``True``, then the values of these weights are
                         sampled from :math:`\mathcal{U}(-\sqrt{k}, \sqrt{k})` where
                         :math:`k = \frac{1}{C_\text{in} * \text{kernel\_size}}`
    """

    def __init__(self, in_channels, out_channels, kernel_size, stride=1,
                 padding=0, output_padding=0, groups=1, bias=True,
                 dilation=1, padding_mode='zeros'):
        kernel_size = _single(kernel_size)
        stride = _single(stride)
        padding = _single(padding)
        dilation = _single(dilation)
        output_padding = _single(output_padding)
        super(ConvTranspose1d, self).__init__(
            in_channels, out_channels, kernel_size, stride, padding, dilation,
            True, output_padding, groups, bias, padding_mode)

    def forward(self, input, output_size=None):
        # type: (Tensor, Optional[List[int]]) -> Tensor
        if self.padding_mode != 'zeros':
            raise ValueError('Only `zeros` padding mode is supported for ConvTranspose1d')

        output_padding = self._output_padding(input, output_size, self.stride, self.padding, self.kernel_size)
        return F.conv_transpose1d(
            input, self.weight, self.bias, self.stride, self.padding,
            output_padding, self.groups, self.dilation)


class ConvTranspose2d(_ConvTransposeMixin, _ConvNd):
    r"""Applies a 2D transposed convolution operator over an input image
    composed of several input planes.

    This module can be seen as the gradient of Conv2d with respect to its input.
    It is also known as a fractionally-strided convolution or
    a deconvolution (although it is not an actual deconvolution operation).

    * :attr:`stride` controls the stride for the cross-correlation.

    * :attr:`padding` controls the amount of implicit zero-paddings on both
      sides for ``dilation * (kernel_size - 1) - padding`` number of points. See note
      below for details.

    * :attr:`output_padding` controls the additional size added to one side
      of the output shape. See note below for details.

    * :attr:`dilation` controls the spacing between the kernel points; also known as the à trous algorithm.
      It is harder to describe, but this `link`_ has a nice visualization of what :attr:`dilation` does.

    * :attr:`groups` controls the connections between inputs and outputs.
      :attr:`in_channels` and :attr:`out_channels` must both be divisible by
      :attr:`groups`. For example,

        * At groups=1, all inputs are convolved to all outputs.
        * At groups=2, the operation becomes equivalent to having two conv
          layers side by side, each seeing half the input channels,
          and producing half the output channels, and both subsequently
          concatenated.
        * At groups= :attr:`in_channels`, each input channel is convolved with
          its own set of filters (of size
          :math:`\left\lfloor\frac{out\_channels}{in\_channels}\right\rfloor`).

    The parameters :attr:`kernel_size`, :attr:`stride`, :attr:`padding`, :attr:`output_padding`
    can either be:

        - a single ``int`` -- in which case the same value is used for the height and width dimensions
        - a ``tuple`` of two ints -- in which case, the first `int` is used for the height dimension,
          and the second `int` for the width dimension

    .. note::

         Depending of the size of your kernel, several (of the last)
         columns of the input might be lost, because it is a valid `cross-correlation`_,
         and not a full `cross-correlation`_.
         It is up to the user to add proper padding.

    .. note::
        The :attr:`padding` argument effectively adds ``dilation * (kernel_size - 1) - padding``
        amount of zero padding to both sizes of the input. This is set so that
        when a :class:`~torch.nn.Conv2d` and a :class:`~torch.nn.ConvTranspose2d`
        are initialized with same parameters, they are inverses of each other in
        regard to the input and output shapes. However, when ``stride > 1``,
        :class:`~torch.nn.Conv2d` maps multiple input shapes to the same output
        shape. :attr:`output_padding` is provided to resolve this ambiguity by
        effectively increasing the calculated output shape on one side. Note
        that :attr:`output_padding` is only used to find output shape, but does
        not actually add zero-padding to output.

    .. include:: cudnn_deterministic.rst

    Args:
        in_channels (int): Number of channels in the input image
        out_channels (int): Number of channels produced by the convolution
        kernel_size (int or tuple): Size of the convolving kernel
        stride (int or tuple, optional): Stride of the convolution. Default: 1
        padding (int or tuple, optional): ``dilation * (kernel_size - 1) - padding`` zero-padding
            will be added to both sides of each dimension in the input. Default: 0
        output_padding (int or tuple, optional): Additional size added to one side
            of each dimension in the output shape. Default: 0
        groups (int, optional): Number of blocked connections from input channels to output channels. Default: 1
        bias (bool, optional): If ``True``, adds a learnable bias to the output. Default: ``True``
        dilation (int or tuple, optional): Spacing between kernel elements. Default: 1

    Shape:
        - Input: :math:`(N, C_{in}, H_{in}, W_{in})`
        - Output: :math:`(N, C_{out}, H_{out}, W_{out})` where

        .. math::
              H_{out} = (H_{in} - 1) \times \text{stride}[0] - 2 \times \text{padding}[0] + \text{dilation}[0]
                        \times (\text{kernel\_size}[0] - 1) + \text{output\_padding}[0] + 1
        .. math::
              W_{out} = (W_{in} - 1) \times \text{stride}[1] - 2 \times \text{padding}[1] + \text{dilation}[1]
                        \times (\text{kernel\_size}[1] - 1) + \text{output\_padding}[1] + 1

    Attributes:
        weight (Tensor): the learnable weights of the module of shape
                         :math:`(\text{in\_channels}, \frac{\text{out\_channels}}{\text{groups}},`
                         :math:`\text{kernel\_size[0]}, \text{kernel\_size[1]})`.
                         The values of these weights are sampled from
                         :math:`\mathcal{U}(-\sqrt{k}, \sqrt{k})` where
                         :math:`k = \frac{1}{C_\text{in} * \prod_{i=0}^{1}\text{kernel\_size}[i]}`
        bias (Tensor):   the learnable bias of the module of shape (out_channels)
                         If :attr:`bias` is ``True``, then the values of these weights are
                         sampled from :math:`\mathcal{U}(-\sqrt{k}, \sqrt{k})` where
                         :math:`k = \frac{1}{C_\text{in} * \prod_{i=0}^{1}\text{kernel\_size}[i]}`

    Examples::

        >>> # With square kernels and equal stride
        >>> m = nn.ConvTranspose2d(16, 33, 3, stride=2)
        >>> # non-square kernels and unequal stride and with padding
        >>> m = nn.ConvTranspose2d(16, 33, (3, 5), stride=(2, 1), padding=(4, 2))
        >>> input = torch.randn(20, 16, 50, 100)
        >>> output = m(input)
        >>> # exact output size can be also specified as an argument
        >>> input = torch.randn(1, 16, 12, 12)
        >>> downsample = nn.Conv2d(16, 16, 3, stride=2, padding=1)
        >>> upsample = nn.ConvTranspose2d(16, 16, 3, stride=2, padding=1)
        >>> h = downsample(input)
        >>> h.size()
        torch.Size([1, 16, 6, 6])
        >>> output = upsample(h, output_size=input.size())
        >>> output.size()
        torch.Size([1, 16, 12, 12])

    .. _cross-correlation:
        https://en.wikipedia.org/wiki/Cross-correlation

    .. _link:
        https://github.com/vdumoulin/conv_arithmetic/blob/master/README.md
    """

    def __init__(self, in_channels, out_channels, kernel_size, stride=1,
                 padding=0, output_padding=0, groups=1, bias=True,
                 dilation=1, padding_mode='zeros'):
        kernel_size = _pair(kernel_size)
        stride = _pair(stride)
        padding = _pair(padding)
        dilation = _pair(dilation)
        output_padding = _pair(output_padding)
        super(ConvTranspose2d, self).__init__(
            in_channels, out_channels, kernel_size, stride, padding, dilation,
            True, output_padding, groups, bias, padding_mode)

    def forward(self, input, output_size=None):
        # type: (Tensor, Optional[List[int]]) -> Tensor
        if self.padding_mode != 'zeros':
            raise ValueError('Only `zeros` padding mode is supported for ConvTranspose2d')

        output_padding = self._output_padding(input, output_size, self.stride, self.padding, self.kernel_size)

        return F.conv_transpose2d(
            input, self.weight, self.bias, self.stride, self.padding,
            output_padding, self.groups, self.dilation)


class ConvTranspose3d(_ConvTransposeMixin, _ConvNd):
    r"""Applies a 3D transposed convolution operator over an input image composed of several input
    planes.
    The transposed convolution operator multiplies each input value element-wise by a learnable kernel,
    and sums over the outputs from all input feature planes.

    This module can be seen as the gradient of Conv3d with respect to its input.
    It is also known as a fractionally-strided convolution or
    a deconvolution (although it is not an actual deconvolution operation).

    * :attr:`stride` controls the stride for the cross-correlation.

    * :attr:`padding` controls the amount of implicit zero-paddings on both
      sides for ``dilation * (kernel_size - 1) - padding`` number of points. See note
      below for details.

    * :attr:`output_padding` controls the additional size added to one side
      of the output shape. See note below for details.

    * :attr:`dilation` controls the spacing between the kernel points; also known as the à trous algorithm.
      It is harder to describe, but this `link`_ has a nice visualization of what :attr:`dilation` does.
```

```

* :attr:`groups` controls the connections between inputs and outputs.
  :attr:`in_channels` and :attr:`out_channels` must both be divisible by
  :attr:`groups`. For example,

    * At groups=1, all inputs are convolved to all outputs.
    * At groups=2, the operation becomes equivalent to having two conv
      layers side by side, each seeing half the input channels,
      and producing half the output channels, and both subsequently
      concatenated.
    * At groups= :attr:`in_channels`, each input channel is convolved with
      its own set of filters (of size
      :math:`\left\lfloor\frac{out\_channels}{in\_channels}\right\rfloor`).

The parameters :attr:`kernel_size`, :attr:`stride`, :attr:`padding`, :attr:`output_padding`
can either be:

    - a single ``int`` -- in which case the same value is used for the depth, height and width dimensions
    - a ``tuple`` of three ints -- in which case, the first `int` is used for the depth dimension,
      the second `int` for the height dimension and the third `int` for the width dimension

.. note::

     Depending of the size of your kernel, several (of the last)
     columns of the input might be lost, because it is a valid `cross-correlation`_,
     and not a full `cross-correlation`_.
     It is up to the user to add proper padding.

.. note::
    The :attr:`padding` argument effectively adds ``dilation * (kernel_size - 1) - padding``
    amount of zero padding to both sizes of the input. This is set so that
    when a :class:`~torch.nn.Conv3d` and a :class:`~torch.nn.ConvTranspose3d`
    are initialized with same parameters, they are inverses of each other in
    regard to the input and output shapes. However, when ``stride > 1``,
    :class:`~torch.nn.Conv3d` maps multiple input shapes to the same output
    shape. :attr:`output_padding` is provided to resolve this ambiguity by
    effectively increasing the calculated output shape on one side. Note
    that :attr:`output_padding` is only used to find output shape, but does
    not actually add zero-padding to output.

.. include:: cudnn_deterministic.rst

Args:
    in_channels (int): Number of channels in the input image
    out_channels (int): Number of channels produced by the convolution
    kernel_size (int or tuple): Size of the convolving kernel
    stride (int or tuple, optional): Stride of the convolution. Default: 1
    padding (int or tuple, optional): ``dilation * (kernel_size - 1) - padding`` zero-padding
        will be added to both sides of each dimension in the input. Default: 0
    output_padding (int or tuple, optional): Additional size added to one side
        of each dimension in the output shape. Default: 0
    groups (int, optional): Number of blocked connections from input channels to output channels. Default: 1
```

```
        bias (bool, optional): If ``True``, adds a learnable bias to the output. Default: ``True``
        dilation (int or tuple, optional): Spacing between kernel elements. Default: 1

    Shape:
        - Input: :math:`(N, C_{in}, D_{in}, H_{in}, W_{in})`
        - Output: :math:`(N, C_{out}, D_{out}, H_{out}, W_{out})` where

        .. math::
              D_{out} = (D_{in} - 1) \times \text{stride}[0] - 2 \times \text{padding}[0] + \text{dilation}[0]
                        \times (\text{kernel\_size}[0] - 1) + \text{output\_padding}[0] + 1
        .. math::
              H_{out} = (H_{in} - 1) \times \text{stride}[1] - 2 \times \text{padding}[1] + \text{dilation}[1]
                        \times (\text{kernel\_size}[1] - 1) + \text{output\_padding}[1] + 1
        .. math::
              W_{out} = (W_{in} - 1) \times \text{stride}[2] - 2 \times \text{padding}[2] + \text{dilation}[2]
                        \times (\text{kernel\_size}[2] - 1) + \text{output\_padding}[2] + 1


    Attributes:
        weight (Tensor): the learnable weights of the module of shape
                         :math:`(\text{in\_channels}, \frac{\text{out\_channels}}{\text{groups}},`
                         :math:`\text{kernel\_size[0]}, \text{kernel\_size[1]}, \text{kernel\_size[2]})`.
                         The values of these weights are sampled from
                         :math:`\mathcal{U}(-\sqrt{k}, \sqrt{k})` where
                         :math:`k = \frac{1}{C_\text{in} * \prod_{i=0}^{2}\text{kernel\_size}[i]}`
        bias (Tensor):   the learnable bias of the module of shape (out_channels)
                         If :attr:`bias` is ``True``, then the values of these weights are
                         sampled from :math:`\mathcal{U}(-\sqrt{k}, \sqrt{k})` where
                         :math:`k = \frac{1}{C_\text{in} * \prod_{i=0}^{2}\text{kernel\_size}[i]}`

    Examples::

        >>> # With square kernels and equal stride
        >>> m = nn.ConvTranspose3d(16, 33, 3, stride=2)
        >>> # non-square kernels and unequal stride and with padding
        >>> m = nn.ConvTranspose3d(16, 33, (3, 5, 2), stride=(2, 1, 1), padding=(0, 4, 2))
        >>> input = torch.randn(20, 16, 10, 50, 100)
        >>> output = m(input)

    .. _cross-correlation:
        https://en.wikipedia.org/wiki/Cross-correlation

    .. _link:
        https://github.com/vdumoulin/conv_arithmetic/blob/master/README.md
    """

    def __init__(self, in_channels, out_channels, kernel_size, stride=1,
                 padding=0, output_padding=0, groups=1, bias=True,
                 dilation=1, padding_mode='zeros'):
        kernel_size = _triple(kernel_size)
        stride = _triple(stride)
        padding = _triple(padding)
        dilation = _triple(dilation)
        output_padding = _triple(output_padding)
        super(ConvTranspose3d, self).__init__(
            in_channels, out_channels, kernel_size, stride, padding, dilation,
            True, output_padding, groups, bias, padding_mode)

    def forward(self, input, output_size=None):
        # type: (Tensor, Optional[List[int]]) -> Tensor
        if self.padding_mode != 'zeros':
            raise ValueError('Only `zeros` padding mode is supported for ConvTranspose3d')

        output_padding = self._output_padding(input, output_size, self.stride, self.padding, self.kernel_size)

        return F.conv_transpose3d(
            input, self.weight, self.bias, self.stride, self.padding,
            output_padding, self.groups, self.dilation)


# TODO: Conv2dLocal
# TODO: Conv2dMap
# TODO: ConvTranspose2dMap
```