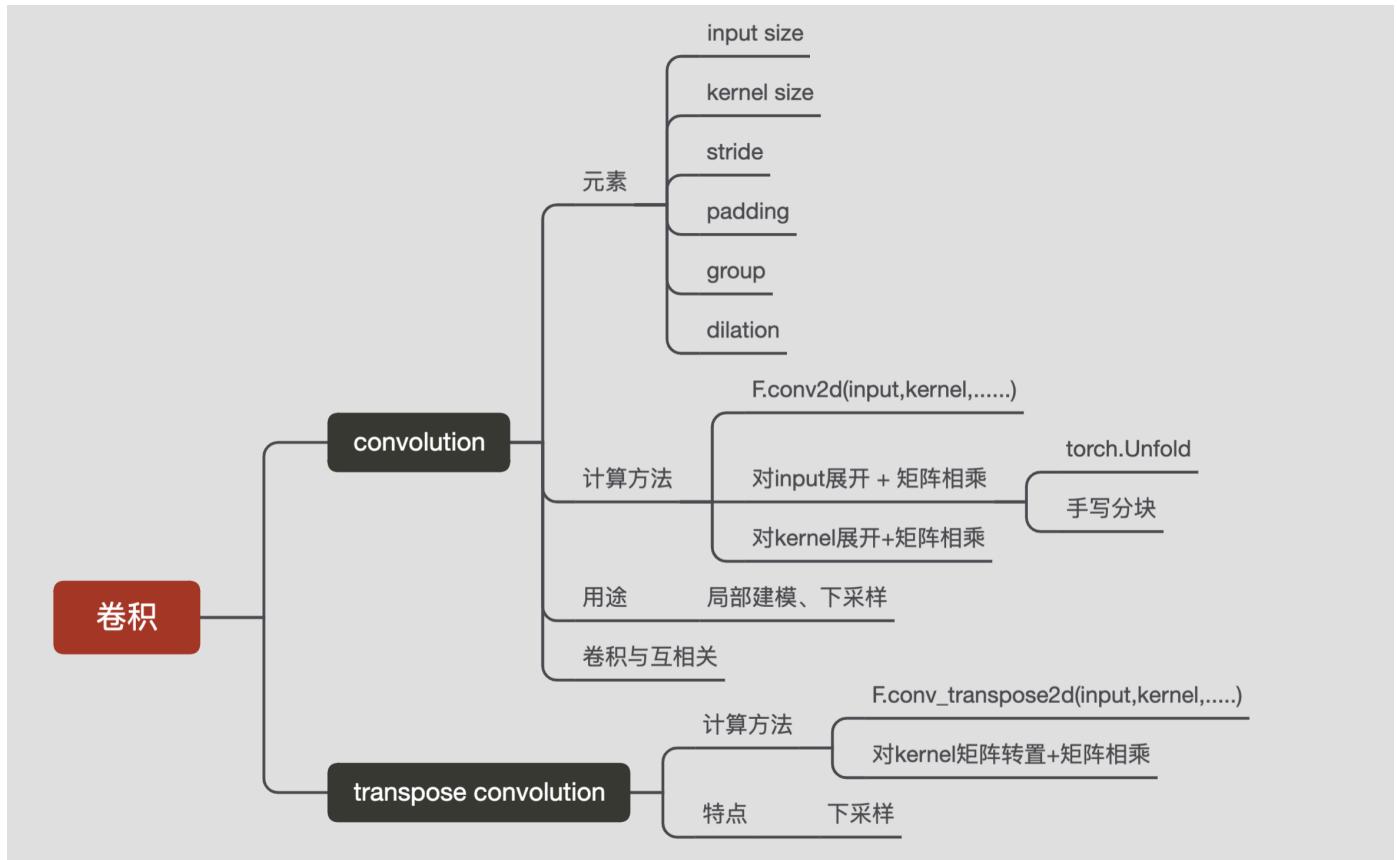


时间线：

- 2024年10月29日
- 2024年10月30日 done

本节框架：



首先 卷积和转置卷积的脑图，3—4次讲；首先 卷积的元素 pytorch中怎么用卷积；之后卷积不一样的计算角度；比方说 把卷积看成是 首先对输入特征图进行展开，然后再进行矩阵的相乘；

另外一种方法 对kernel或者filter进行展开，然后再进行矩阵相乘，有了这种方法 可以顺其自然的引出 转置卷积；之后会讲 转置卷积 也有人称为反卷积，但是反卷积的说法不太准确，因为 转置卷积虽然说是上采样，但是不能从output去恢复input，转置卷积 恢复的只是 input的形状，不是input的元素值；更准确的定义 就是转置卷积；为什么叫转置卷积呢？再讲完 对kernel 进行展开，再进行矩阵相乘 就可以明白了；

当我们把常规的卷积 看成是对kernel的展开，然后再矩阵相乘的话，那么转置卷积可以看成将kernel进行一个 转置操作，然后再进行矩阵相乘，就能得到转置卷积的输出

首先讲卷积的基础部分，首先卷积分为一维卷积和二维卷积，pytorch中也是分为conv1d&conv2d,一维卷积可以用二维卷积做，着重讲解二维卷积，一维卷积可以通过二维卷积实现[到底什么是以卷积？]

首先看一下二维卷积的api

Google search results for "pytorch conv2d":

- <https://pytorch.org/torch.nn.Conv2d.html> 翻译此页
Conv2d — PyTorch 1.10.0 documentation
Applies a 2D convolution over an input signal composed of several input planes. ... where \star is the valid 2D cross-correlation operator, $N \times N$ is a batch ...
- <https://pytorch.org/docs/stable/generated/torch.nn.functional.conv2d.html> 翻译此页
torch.nn.functional.conv2d — PyTorch 1.10.0 documentation
Applies a 2D convolution over an input image composed of several input planes. This operator supports TensorFloat32. See [Conv2d](#) for details and output shape.

谷歌搜索 pytorch conv2d，出现两个api 一个是大写的二维卷积、一个是torch.nn.functional.conv2d这个小写的二维卷积；如果对pytorch熟悉的话，会知道区别；

首先，第一个大写的是一个class，如果我们要用第一个的话，我们首先需要对这个class进行一个实例化，然后对实例化的对象，再对输入特征图进行一个卷积操作；

第二个是一个函数，不需要实例化，就直接接收一个输入特征图，直接进行一个卷积操作；以上是第一个区别；

第二个区别就是，class可以自己去创建操作，包括weight和bias，可以自动去创建，就不需要手动创建；

对于函数来说，需要手动的传入weight和bias；

接下来看class的这个api，也就是大写的CONV2D

1.10.0 ▾

Search Docs

Notes
Automatic Mixed Precision examples
Autograd mechanics
Broadcasting semantics
CPU threading and TorchScript inference
CUDA semantics
Distributed Data Parallel
Extending PyTorch
Frequently Asked Questions
Gradcheck mechanics
HIP (ROCM) semantics
Features for large-scale deployments
Modules
Multiprocessing best practices
Reproducibility
Serialization semantics
Windows FAQ

Language Bindings

CONV2D

Conv2d

CLASS `torch.nn.Conv2d(in_channels, out_channels, kernel_size, stride=1, padding=0, dilation=1, groups=1, bias=True, padding_mode='zeros', device=None, dtype=None)` [SOURCE]

Applies a 2D convolution over an input signal composed of several input planes.

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

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

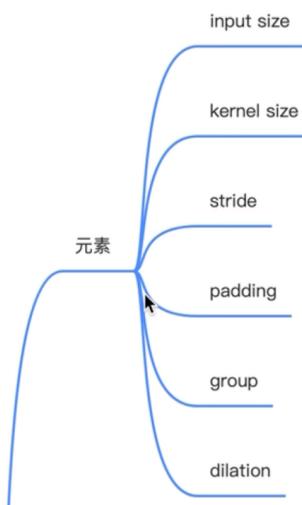
where $*$ is the valid 2D cross-correlation operator, N is a batch size, C denotes a number of channels, H is a height of input planes in pixels, and W is width in pixels.

This module supports `TensorFloat32`.

- `stride` controls the stride for the cross-correlation, a single number or a tuple.
- `padding` controls the amount of padding applied to the input. It can be either a string {'valid', 'same'} or a tuple of ints giving the amount of implicit padding applied on both sides.
- `dilation` controls the spacing between the kernel points; also known as the à trous algorithm. It is harder to describe, but this link has a nice visualization of what `dilation` does.

首先大写的api也是在torch.nn这个模块下面，调用torch.nn.Conv2d就好了；我们可以看到如果要实例化一个二维卷积的操作的话，需要传入的参数：输入通道、输出通道、kernel的大小、步长、padding填充、膨胀dilation、group；

今天主要讲解in channels、out channels、kernel size、stride、padding、bias；dilation、group之前讲过：残差网络的算子融合

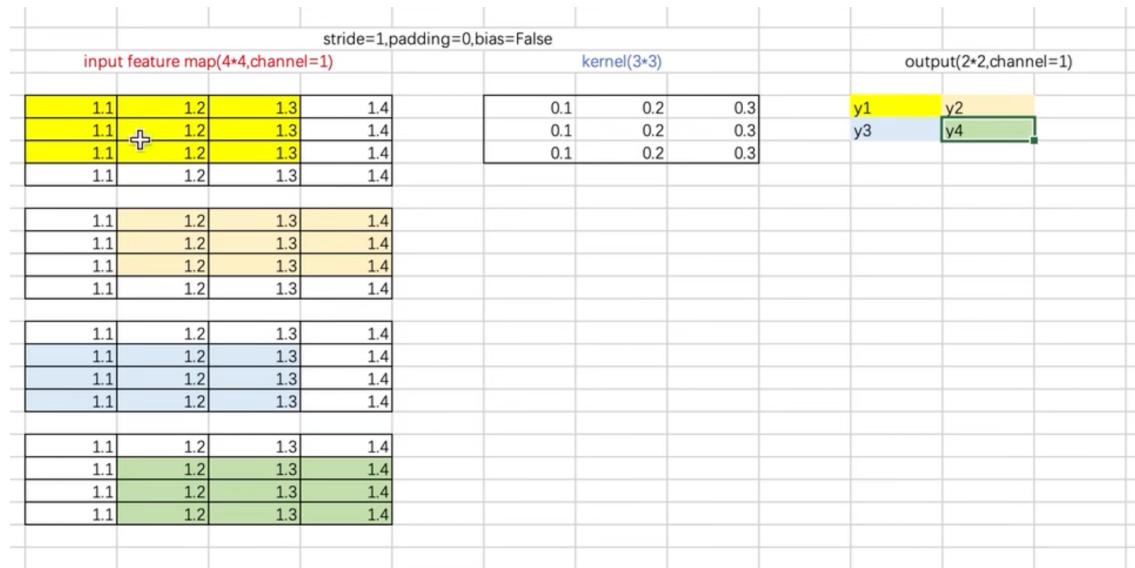


首先明确卷积和全连接网络的区别是什么，为什么会有卷积的操作。

神经网络最核心的一个操作：仿射变换：将一个矩阵乘以输入向量得到另外一个向量。这是全连接网络的一个做法，所以我们一般会对一个向量做全连接的网络的输入；比方说：一个word embedding向量；比方说要预测房价，城市的人口还有物价等，不同的浮点数组成的向量，这些都可以送入全连接网络。

所以全连接网络是把输入当成一个向量，然后统一的去乘一个矩阵，进行操作。但是，还有很多其他东西，不能仅仅使用一个向量来进行刻画，比如图像有长度和宽度，是一个二维的，还有RGB三个通道，这些我们不能仅仅只是把图片拉直处理，这样破坏了图片的空间结构；

类似的还有语音，语言有时间维还有频率维，我们每个时刻发出的声音，是由不同的频率组合的，同样对于语音这种信号，我们也不能仅仅是当成一维信号处理，甚至更复杂的是图像和语音信号的结合，比如视频。所以对于这些我们不能仅仅只是当成一个向量处理，这样的话，全连接网络也就无法刻画它，我们可以用卷积网络刻画，对于卷积网络和哪些操作比较相关呢？就是互相关，如果学过信号与系统的话，互相关就是对于两个一维向量，我们把一个一维信号沿着另外一个一维信号，不断地进行滑动相乘的操作，然后计算一个相关系数。卷积也是类似的，对于一张图片，如果我们有一个卷积核的话，叫做kernel，我们会把kernel沿着图片的不同区域进行一个滑动相乘，来得到一个特征的表示，接下来excel演示卷积的具体操作：



- 假设我们的input feature map= 4×4 , kernel= 3×3 , 卷积操作就是将kernel在图片上不同位置元素相乘 element-wise, 不同位置元素相乘再相加, 得到输出;
 - k=3, p=0, s=1
 - kernel的移动轨迹是Z字型的, 从左到右, 从上到下
 - 输入input feature map的大小是 4×4 的, 而且 channel=1, 再用一个 3×3 的kernel, 与输入特征图进行卷积操作, 得到output, 并且output大小 2×2 , channel=1, 同时这里我们设置的bias=False, 不加 bias;
 - 如果我们加入 bias呢?
 - 如果 channel=1, 那么 bias就是一个标量, 直接相加就好了, 这就是一个 bias的操作

- 如果 输入的通道数不止是1呢？比如两个通道，这个时候 就会有两个kernel，第一个 kernel得到y1 y2 y3 y4；第二个kernel又会得到一个y1, y2,y3,y4,然后我们再把两个 kernel得到的输出 再进行一个点对点的输出，这样得到 最终的output，这是对输入特征图有多个通道的情况。
- 那如果我们 输出 特征图 也有多个通道的情况 会怎么处理呢？刚刚 我们得到了第一个通道，对于第二个通道，我们同样 在另外创造 不同的kernel，对输入进行一个卷积操作，最后把 输入的通道 加起来，变成 输出 通道的第二个输出
- 以上是所有 卷积的过程：
 - 有几个卷积核 就有几个 输出通道；
 - 单个卷积核的通道数 取决于 输入特征图的通道数
- 我们将 3×3 的kernel，在输入的特征图上 进行一个Z字型的滑动相乘的操作
 - 其实这里的滑动相乘 可以理解为 如果把输入的特征图（被卷积核覆盖的区域） 3×3 的区域 拉成一个向量的话 然后我们把kernel也拉成一个向量，其实 就是计算 两个向量的一个内积。内积越大 两个向量 越相似。
- 所以卷积网络 学习的是什么呢？卷积网络 会 不断的更新 kernel和 bias。就是为了学到：
 - 比方说 人脸识别，就希望kernel能够学到 能够反映人脸的 特征，然后把kernel 对图片的不同区域，进行比对，如果刚好发现，图片的某一个区域刚好与人脸的 kernel很相似的话，那就说明你给我们已经找到人脸了，总之卷积神经网络是 给定一个目标 不断的学习kernel，最终希望kernel，能够跟图片的某一个区域，相似度达到一个比较高的值，得到一个比较好的特征，然后再不断的往 深层去传
- 以上是 卷积网络的演示，接下来 pytorch中 调用 卷积网络，演示二维卷积，首先导入 torch,torch.nn

```
import torch
import torch.nn as nn
import torch.nn.functional as F
```

- 首先看class的api

```
conv_layer = torch.nn.Conv2d()
```

CONV2D

```
CLASS torch.nn.Conv2d(in_channels, out_channels, kernel_size, stride=1, padding=0,
dilation=1, groups=1, bias=True, padding_mode='zeros', device=None, dtype=None) [SOURCE]
```

Applies a 2D convolution over an input signal composed of several input planes.

第一个位置需要传入 输入通道，也就是说 输入的feature map有几个通道，对于一张RGB，一般有三个通道，演示 假设我们有1个通道。

输出通道数 意思是我们希望 将 输入特征图 映射到几个通道上，为演示 我们也可以设置为1，我们可以首先 定义几个常量，通过常量传入参数

第三个参数 kernel size,定义为3，这里可以传入标量（表示方阵），可以传入元组（表示矩形，宽高 可以不同）

padding默认为0， stride默认为1

bias默认为True， 我们演示改成False

这样 我们实例化了一个conv layer

```
in_channels = 1
out_channels = 1
kernel_size = 3
bias = False

conv_layer =
torch.nn.Conv2d(in_channels, out_channels, kernel_size, bias=bias)
```

实例化好一个 conv layer以后，需要 传入一个 input

卷积需要有一个 input，用kernel在input上进行 滑动相乘，得到最终的一个输出，所以我们还需要定义一个input size ，也就是 除了 通道维以外，这个宽度 和 高度分别是多少，首先 通道数 设为 in_channels，宽度和高度 分别设为4和4

```
input_size = [in_channels, 4, 4]
```

接下来 通过torch.randn,一个 高斯分布函数，产生一个 input,传入 input_size即可

```
input_feature_map = torch.randn(input_size )
```

然后 我们把 input_feature_map,作为卷积层的输入 就好了

```
output_feature_map = conv_layer(input_feature_map)
```

这样 就得到了 output feature map

```
In [1]: import torch
import torch.nn as nn
import torch.nn.functional as F
in_channels = 1
out_channels = 1
kernel_size = 3
bias = False
input_size = [in_channels, 4, 4]

conv_layer = torch.nn.Conv2d(in_channels, out_channels, kernel_size, bias=bias)
input_feature_map = torch.randn(input_size)
output_feature_map = conv_layer(input_feature_map)

-----
RuntimeError
Traceback (most recent call last)
/var/folders/jm/0msk3gg922j555fq8185txch000gp/T/ipykernel_90419/3402741698.py in <module>
    10 conv_layer = torch.nn.Conv2d(in_channels, out_channels, kernel_size, bias=bias)
    11 input_feature_map = torch.randn(input_size)
--> 12 output_feature_map = conv_layer(input_feature_map)

/usr/local/lib/python3.9/site-packages/torch/nn/modules/module.py in _call_impl(self, *input, **kwargs)
    1049         if not (self._backward_hooks or self._forward_hooks or self._forward_pre_hooks or _global_backward_hooks
  oks
    1050             or _global_forward_hooks or _global_forward_pre_hooks):
-> 1051     return forward_call(*input, **kwargs)
    1052     # Do not call functions when jit is used
    1053     full_backward_hooks, non_full_backward_hooks = [], []

/usr/local/lib/python3.9/site-packages/torch/nn/modules/conv.py in forward(self, input)
    441
    442     def forward(self, input: Tensor) -> Tensor:
--> 443         return self._conv_forward(input, self.weight, self.bias)
    444
    445 class Conv3d(_ConvNd):

/usr/local/lib/python3.9/site-packages/torch/nn/modules/conv.py in _conv_forward(self, input, weight, bias)
    437             weight, bias, self.stride,
```

运行会报错，因为 Conv2d默认输入是4维的，第一维是batch size维，我们设置batch size=1，并添加到input_size即可；

input feature map的形状： **batch size × 通道数 × 高 × 宽** 可以查看官网 找到需要的输入形状

Convd – PyTorch 2.5 docum +

docs/stable/generated/torch.nn.Conv2d.html

Docs > torch.nn > Conv2d

Shortcuts

- **stride** (*int or tuple, optional*) – Stride of the convolution. Default: 1
- **padding** (*int, tuple or str, optional*) – Padding added to all four sides of the input. Default: 0
- **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`
- **padding_mode** (*str, optional*) – ‘zeros’, ‘reflect’, ‘replicate’ or ‘circular’. Default: ‘zeros’

Shape:

- Input: $(N, C_{in}, H_{in}, W_{in})$ or (C_{in}, H_{in}, W_{in})
- Output: $(N, C_{out}, H_{out}, W_{out})$ or $(C_{out}, H_{out}, W_{out})$, where

$$H_{out} = \left\lfloor \frac{H_{in} + 2 \times \text{padding}[0] - \text{dilation}[0] \times (\text{kernel_size}[0] - 1) - 1}{\text{stride}[0]} + 1 \right\rfloor$$
$$W_{out} = \left\lfloor \frac{W_{in} + 2 \times \text{padding}[1] - \text{dilation}[1] \times (\text{kernel_size}[1] - 1) - 1}{\text{stride}[1]} + 1 \right\rfloor$$

运行不报错了

```
In [2]: import torch
import torch.nn as nn
import torch.nn.functional as F
in_channels = 1
out_channels = 1
kernel_size = 3
batch_size = 1
bias = False
input_size = [batch_size, in_channels, 4, 4]

conv_layer = torch.nn.Conv2d(in_channels, out_channels, kernel_size, bias=bias)
input_feature_map = torch.randn(input_size)
output_feature_map = conv_layer(input_feature_map)
```

我们可以打印一下卷积层的 weight, 也就是kernel, 还可以打印输入和输出

```
In [3]: import torch
import torch.nn as nn
import torch.nn.functional as F
in_channels = 1
out_channels = 1
kernel_size = 3
batch_size = 1
bias = False
input_size = [batch_size, in_channels, 4, 4]

conv_layer = torch.nn.Conv2d(in_channels, out_channels, kernel_size, bias=bias)
input_feature_map = torch.randn(input_size)
output_feature_map = conv_layer(input_feature_map)
print(input_feature_map)
print(conv_layer.weight)
print(output_feature_map)

tensor([[[[-0.3508,  1.9242,  0.8938, -0.2322],
          [ 0.2797, -2.1567, -1.4089, -0.7050],
          [ 1.0373,  0.6111, -0.0473,  1.2506],
          [-0.2059,  0.0521, -0.0578, -0.2146]]]])
Parameter containing:
tensor([[[[ 0.2372,  0.0935,  0.3123],
          [-0.0540,  0.2891, -0.1263],
          [-0.0857,  0.1702, -0.1640]]]], requires_grad=True)
tensor([[[-6.2029e-02,  1.0302e-04],
          [-4.1261e-01, -1.0472e+00]]]], grad_fn=<ThnnConv2DBackward>)
```

输出三个张量 第一个是 输入特征图、第二个是卷积的weight、或者kernel，第三个是 卷积的输出

输出的大小是 $1 \times 1 \times 4$ 的；

kernel是 $1 \times 1 \times 3 \times 3$ 权重就是out channel \times input channel \times height \times width

也就是说对于二维卷积，weight是4维的，那么总的数目 等于 输出通道数 \times 输入通道数 \times 卷积核的高度 \times 卷积核的宽度，如果我们认为 卷积核是一个二维的图片的话，那么一共有 输入通道数 \times 输出通道数 这么多个 卷积核图片

那换句话说 比如说 输入通道数是3，输出通道数 也是3的话，那也就是说一共有9个 kernel，这9个kernel的作用是不一样的，就是说对于第一个输出通道，有三个 kernel 跟输入的三个通道 进行 卷积操作，然后求和。对于输出通道的 第二个通道 也有三个kernel 分别对输入的 三个通道 进行一个卷积；对于输出的第 三个通道 同理。所以一共是 9个二维的 kernel；

```
In [3]: import torch
import torch.nn as nn
import torch.nn.functional as F
in_channels = 1
out_channels = 1
kernel_size = 3
batch_size = 1
bias = False
input_size = [batch_size, in_channels, 4, 4]

conv_layer = torch.nn.Conv2d(in_channels, out_channels, kernel_size, bias=bias)
input_feature_map = torch.randn(input_size)
output_feature_map = conv_layer(input_feature_map)
#print(input_feature_map)
#print(conv_layer.weight) # 1*1*3*3=out_channels*in_channels*height*width
print(output_feature_map)

tensor([[[[-0.3508,  1.9242,  0.8938, -0.2322],
          [ 0.2797, -2.1567, -1.4089, -0.7050],
          [ 1.0373,  0.6111, -0.0473,  1.2506],
          [-0.2059,  0.0521, -0.0578, -0.2146]]]])
Parameter containing:
tensor([[[[ 0.2372,  0.0935,  0.3123],
          [-0.0540,  0.2891, -0.1263],
          [-0.0857,  0.1702, -0.1640]]]], requires_grad=True)
tensor([[[[-6.2029e-02,  1.0302e-04],
          [-4.1261e-01, -1.0472e+00]]]], grad_fn=<ThnnConv2DBackward>)
```

以上是通过torch.nn.Conv2d得到的结果

接下来 我们用 functional的api跑一下 流程

1.10.0 ▾

Docs > torch.nn.functional > torch.nn.functional.conv2d

TORCH.NN.FUNCTIONAL.CONV2D

`torch.nn.functional.conv2d(input, weight, bias=None, stride=1, padding=0, dilation=1, groups=1) → Tensor`

Applies a 2D convolution over an input image composed of several input planes.

This operator supports `TensorFloat32`.

See `Conv2d` for details and output shape.

• NOTE

In some circumstances when given tensors on a CUDA device and using CuDNN, this operator may select a nondeterministic algorithm to increase performance. If this is undesirable, you can try to make the operation deterministic (potentially at a performance cost) by setting `torch.backends.cudnn.deterministic = True`. See [Reproducibility](#) for more information.

Parameters

对于这个api 我们需要手动的指定 weight 和 bias，为了验证，我们可以直接把刚刚的 weight传入，可以看到结果是一样的

```
output_feature_map1 = F.conv2d(input_feature_map, conv_layer.weight)
```

```
: import torch
import torch.nn as nn
import torch.nn.functional as F
in_channels = 1
out_channels = 1
kernel_size = 3
batch_size = 1
bias = False
input_size = [batch_size, in_channels, 4, 4]

conv_layer = torch.nn.Conv2d(in_channels, out_channels, kernel_size, bias=bias)
input_feature_map = torch.randn(input_size)
output_feature_map = conv_layer(input_feature_map)
#print(input_feature_map)
#print(conv_layer.weight) # 1*1*3*3=out_channels*in_channels*height*width
print(output_feature_map)

output_feature_map1 = F.conv2d(input_feature_map, conv_layer.weight)
print(output_feature_map1)

tensor([[[[ 1.7119,  0.5139],
          [-0.6549, -0.3206]]]], grad_fn=<ThnnConv2DBackward>)
tensor([[[[ 1.7119,  0.5139],
          [-0.6549, -0.3206]]]], grad_fn=<ThnnConv2DBackward>)
```

kernel就是在训练中，不断更新的

以上演示了两个api的相同与不同

其实在 class 的 api 中，调用的也是 functional 的 api，functional 的 api 调用的是 pytorch C++ 代码，这个代码很复杂而且分 CPU 和 GPU

本节课所有代码：

```
import torch
import torch.nn as nn
import torch.nn.functional as F

in_channels = 1
out_channels = 1
kernel_size = 3
batch_size = 1
bias = False

input_size = [batch_size, in_channels, 4, 4]

# 第一种实现
conv_layer =
torch.nn.Conv2d(in_channels, out_channels, kernel_size, bias=bias)

input_feature_map = torch.randn(input_size)
out_feature_map = conv_layer(input_feature_map)
# print(input_feature_map)
# print(conv_layer.weight) #
1*1*3*3=out_channels*in_channels*height*width

print(out_feature_map)

out_feature_map1 = F.conv2d(input_feature_map, conv_layer.weight)

print(out_feature_map1)
```