

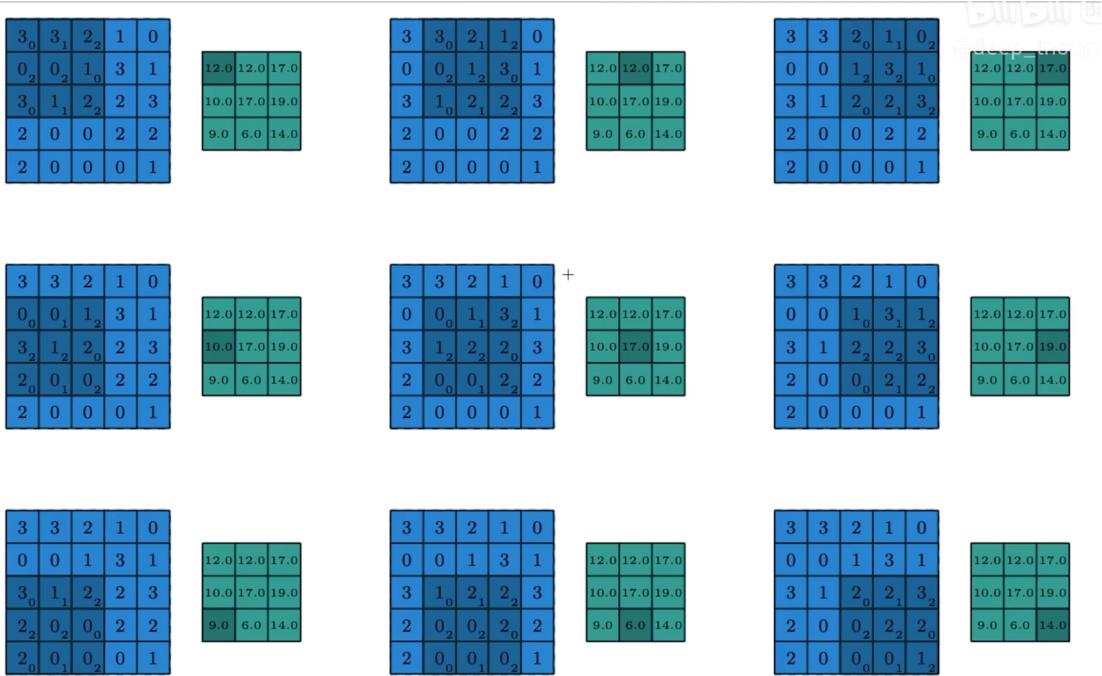
【23、手写并验证滑动相乘实现PyTorch二维卷积-哔哩哔哩】 <https://b23.tv/Jq2s23a>

## 时间线：

- 2024年10月30日 start
  - 2024年10月31日 done

上次直播讲的是卷积的两个api的介绍，分别是一个函数api和一个class的api，也从代码上将两者的结果进行了验证，两种实现方式殊途同归的可以实现同样的计算结果，同时也讲了卷积的基本原理，接下来继续讲解卷积的原理，以及从矩阵运算的角度实现二维卷积，并且跟框架的结果进行对比验证，今天代码为主。

接下来 回顾一下卷积的原理

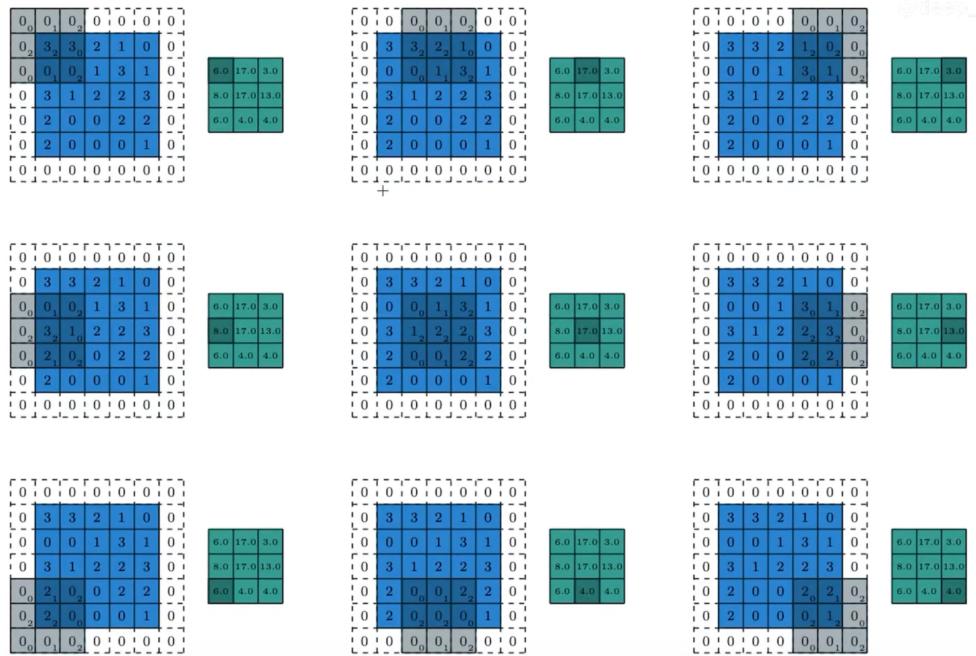


- 首先这个蓝色的大的张量是input feature map, 也就是输入特征图
  - 这个绿色的 $3 \times 3$ 的特征图, 就是output feature map,
  - 图中有9个示意图, 分别演示了kernel在feature map上滑动的过程, 从左到右 从上到下, 这个 $3 \times 3$ 的 kernel, 依次的从feature map左上角 进行一个点对点的运算,
  - feature map的数 分别是332、001、312, kernel是012、220、012; 左上角 kernel与feature map左上角的区域 进行 卷积运算, 逐元素相乘 再求和 得到 输出的12;
  - 接下来 把feature map向右 移动一个单位, 我们这里 默认stride=1, 同样进行 逐元素的相乘, 得到12, 这就是 feature map的第二个数
  - 同理, 依次这么做, 最终移动9次, 完成卷积操作
  - 上图 padding=0, stride=1
  - 卷积有哪些元素呢?
    - 第一个input feature map的大小, 这里的feature map是一个 $5 \times 5$ 的

- 第二个元素是kernel的大小, kernel的大小是 $3 \times 3$ 的kernel
- 第三个元素是移动的步长, 也就是stride, 我们这里stride=1
- 第四个元素是padding, 在本图中, padding可以理解为0, 意思就是说并没有对图片进行一个pad
- 第五个元素是 channel, 只是这里的设置输入channel和输出channel都是1, 等一下看到另外一张图是多channel的情况

以上是第一幅图,

现在来看第二幅图



第二张图稍微复杂一点, 这里我们改变了两个元素,

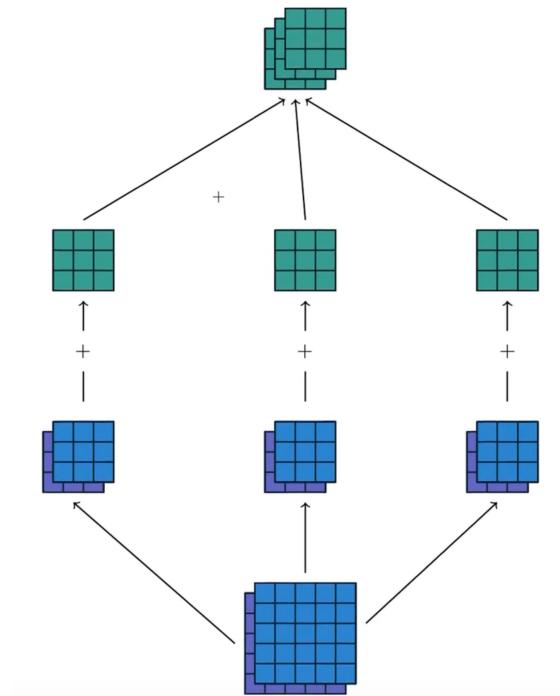
- 第一个元素是对输入的input feature map做了一个padding操作, 可以看到pad的大小是1, 比方说我们这里用到的是一个二维的图像, 所以我们是对上方下方左方右方都pad了一个步长; [什么是一维卷积? ]所以可以看到本来是一个 $5 \times 5$ 的feature map变成了 $7 \times 7$ 的input feature map[pytorch中pad的参数是什么意思? ]
- 第二个改变的地方, stride步长从1改成了2, 可以看到第一幅图从左上角开始, 仍然是 $3 \times 3$ 的kernel, 但到第二幅图的话, 就是往右移动了两个步长, 所以这里stride是等于2的, 然后呢第三幅图的时候我们继续移动了两个步长, 刚好到边界了, 所以第四幅图我们就把channel, 向下也是移动两个步长, stride也就是两个维度, 都是2, 所以往下也是移动两个步长, 同样进行向量的内积操作, 得到了8.0;
- 但是这幅图与刚刚那副图有什么相同就是呢我们输出的feature map大小都是 $3 \times 3$ 的这样的feature map, 为什么呢?
  - 因为虽然我们对feature map进行了一个padding, 上下左右都是pad 1, 但是我们的步长其实是扩大了, 从1变成了2, 所以如果我们的步长仍然设置为1的话, 那么我们这里输出的图大小是多少呢?  

$$5 - 3 + 2 + 1 = 5$$
 即输出 $5 \times 5$ 的feature map, 但我们把stride设置成了2,  

$$\frac{n-k+1+2p+s}{s} = \frac{5-3+2+2}{2} = 3$$

以上是第二幅图的讲解 但这里input channel和output channel都是1, 这种情况比较简单, 但我们实际用的不是这样的, 在我们实际使用的过程中通道通常是比较大的input通常是3, 卷积层之后的输出通道数会更多

接下来 第三幅图



展示的是 多通道的情况,

- 首先 最下面时 input feature map, 最上方是 output feature map, 可以看到input feature map 的通道数等于2, channel=2, 对应着 convolution 的api 就是 in channels
- 再看 最上面的 output feature map, 有三个map, 也就是 这里的 out channels=3
- 首先 kernels[这里讲的有点怪 但是也能理解 也还好]的数目怎么算, 是 $\text{in\_channels} \times \text{out\_channels}$ 
  - 这里 in channels=2,out channels=3,所以 kernels=6
  - 对应图中 第二行 表示 kernel
- 现在解决 这6个kernel 是怎么分配的
  - 因为我们输入是2通道 输出是3通道
  - [卷积核数量决定输出特征图的channel; 卷积核的channel取决于 输入特征图的channel]
  - 输出的第一个通道的结果 由 输入的每个 通道, 分别与 各自的 每个 kernel, 进行一个 卷积操作, 进行我们刚刚的最简单的 卷积操作, 得到了 两个  $3 \times 3$ 的 map, 然后 我们再把 两个  $3 \times 3$ 的map, 加起来, 构成了 输出 通道的第一个map
  - 对于 输出的 第二个通道 map同样, 由两个kernel跟输入特征的两个通道 分别进行卷积, 然后 再把 这两个卷积的结果 进行相加 (element-wise), 得到 输出 特征图的 第二个channel
  - 同样, 输出特征图的第三个channels, 是由另外 两个kernel 跟 输入特征的 两个map 进行卷积操作, 最后相加 得到输出特征图的 第三个map

关于输出特征图的几点讨论, 尺寸  $\frac{n-k+s+2p}{s}$

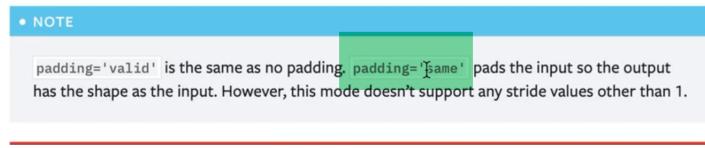
要满足 输入特征图 和输出特征图 尺寸相同 则令  $\frac{n-k+s+2p}{s} = n$ , 则

$$n - k + s + 2p = sn, (s - 1)n = s + 2p - k$$

$s = 1$ 时,  $k = s + 2p$ 即可,  $s$ 默认等于1

$s \neq 1$ 时,  $n = \frac{s+2p-k}{s-1}$

pytorch的api提供了: `same` padding



## 代码

利用矩阵运算演示卷积的流程, 上次讲的两个api的调用

```
: 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) # I1*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>)
```

这里有input feature map和conv\_layer.weight, 我们把这两个单独拎出来

```
input = input_feature_map # 卷积输入特征图
kernel = conv_layer.weight.data #卷积核
# step1 用原始的矩阵运算来实现二维卷积
```

, 用原始的矩阵运算实现一个二维卷积, 首先 定义一个函数 `matrix_multiplication_for_conv2d`, 输入是 input、kernel、stride默认为1, padding默认为0, 但我们暂时先不设置

```
#step1 用原始的矩阵运算来实现二维卷积
def matrix_multiplication_for_conv2d(input, kernel, stride=1):
```

有输入、有kernel、有stride, padding暂时没写, 用原始的矩阵运算 实现 二维卷积

首先 获取一个大小 input大小 和 kernel大小

为了简单一点 先以矩形 举例，先不考虑 通道数，随机生成一个 input，设为 $5 \times 5$ ，kernel设为 $3 \times 3$ ，这些都没考虑 batch size维度 和 channel维度

```
input = torch.randn(5,5) # 卷积 输入特征图
kernel = torch.randn(3,3) # 卷积核
# step1 用原始的矩阵运算来实现二维卷积，先不考虑 batch size维度 和 channel维度
def matrix_multiplication_for_conv2d(input,kernel,stride=1):
```

首先得到 input的高度和宽度，同样 kernel也是一样：

```
input = torch.randn(5,5) # 卷积 输入特征图
kernel = torch.randn(3,3) # 卷积核
# step1 用原始的矩阵运算来实现二维卷积，先不考虑 batch size维度 和 channel维度
def matrix_multiplication_for_conv2d(input,kernel,stride=1):
    input_h,input_w = input.shape
    kernel_h,kernel_w = kernel.shape
```

然后进行循环，首先 搞出 output的形状，先计算output的宽度，output的宽度 应该是等于 input的宽度 减去 kernel的宽度 加上1，当然 这是 stride= 1 的情况

$$\frac{n-k+s+2p}{s}$$

这边 我们用 一个floor函数，为什么呢？我们这里 第一步，第一步 kernel放在这里

3	0	3	2	1	0
0	2	0	1	3	1
3	1	2	2	3	
2	0	0	2	2	
2	0	0	0	1	

还剩下 多少距离呢？ $n - k$ 个距离， $n - k$ 个距离我们要走几次呢？因为 我们每次 步长是stride，应该是  
 $\frac{n-k+stride}{stride}$

这样 得到 输出的宽度

```
input = torch.randn(5,5) # 卷积 输入特征图
kernel = torch.randn(3,3) # 卷积核
# step1 用原始的矩阵运算来实现二维卷积，先不考虑 batch size维度 和 channel维度
def matrix_multiplication_for_conv2d(input,kernel,stride=1):
    input_h,input_w = input.shape
    kernel_h,kernel_w = kernel.shape

    output_h = (floor(input_h - kernel_h)/stride + 1) # 卷积输出的高度
    output_w = (floor(input_w - kernel_w)/stride + 1) # 卷积输出的宽度
```

输出 高度呢？也是类似的，输出的高度 就是 输入的高度 减去kernel的高度 在除以 stride，我们 默认 stride 是宽度 高度的stride

得到卷积输出的高度 和宽度以后 随机初始化一个 输出矩阵，就是卷积的输出大小

```
output = torch.zeros(output_h,output_w) # 初始化 输出矩阵
```

卷积的输出大小 跟 输入的大小、kernel的大小、stride的有关，还有padding

接下来 我们来进行 遍历，我们这里 进行两层遍历

首先是 高度维 和 宽度维，我们以i为高度维 进行遍历 for i in range(0,)从0开始 到哪里 结束呢？到边界结束，最左边 就是 kernel的最右边刚好和 input的最右边 相重合，具体地用公式怎么写呢？

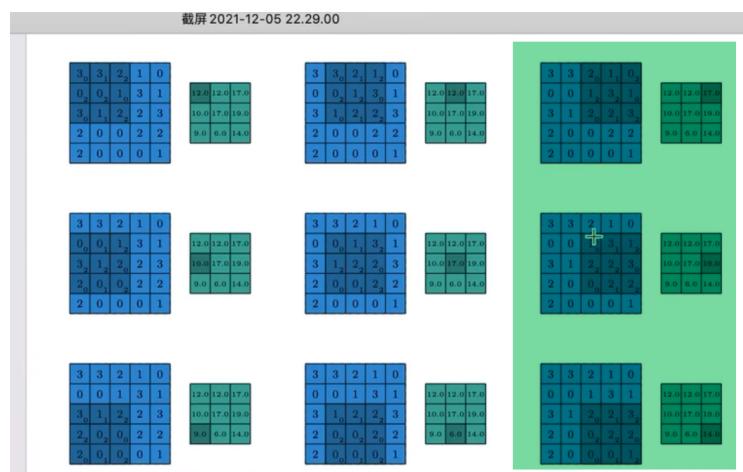
就是 `input_h - kernel_h + 1` 也就是 input 的最底边 与 kernel的最底边 相重合，也就是 input的高度 -kernel 的高度 + 1，这就是高度循环的边界，那高度循环的 步长为多少呢？就是stride，这就是 对高度维 进行遍历

```
for i in range(0,input_h - kernel_h + 1,stride): # 对高度进行遍历
```

里面还有一层循环，还有一个 对宽度维 进行遍历 同样 宽度 也是一样的 `input_w - kernel_w + 1`，步长也是 stride

这就是在输入的特征图上 进行 位置遍历，第一时刻 在第 0 列，第二 时刻 就是在 stride这一列，第三时刻 就是 2stride这一列，一直到我们触碰到 `input_w - kernel_w +1`

```
for i in range(0,input_h - kernel_h + 1,stride): # 对高度进行遍历  
    for j in range(0,input_w - kernel_w +1,stride): # 对宽度维进行遍历
```



`input_w - kernel_w +1`就是 上图 这一列

遍历完 以后 可以对矩阵 进行一个 相乘了 就是 矩阵 逐元素的相乘，这就比较简单了[难死了 谢谢]

首先 我们取一个区域 region,区域 就是 我们输入的区域，高度 就是i到i+kernel\_h,这就是高度维，那宽度维就是j,j 到 j+ kernel\_w,这就是我们现在 得到的 input区域，

```
for i in range(0,input_h - kernel_h + 1,stride): # 对高度进行遍历  
    for j in range(0,input_w - kernel_w +1,stride): # 对宽度维进行遍历  
        region = input[i:i+kernel_h, j:j+kernel_w]
```

然后呢，我们把 region和kernel 进行一个逐元素的相乘，然后再调用一个 torch.sum函数，得到当前位置的输出，我们把输出 赋值给output即可

```
for i in range(0,input_h - kernel_h + 1,stride): # 对高度进行遍历
    for j in range(0,input_w - kernel_w +1,stride): # 对宽度维进行遍历
        region = input[i:i+kernel_h, j:j+kernel_w]
        output[] = torch.sum(region * kernel)
```

那么这里 output的索引 怎么写呢？output就是高度 i/stride， 宽度就是 j/stride, 我们还要用一个 int 转成一个整型

```
for i in range(0,input_h - kernel_h + 1,stride): # 对高度进行遍历
    for j in range(0,input_w - kernel_w +1,stride): # 对宽度维进行遍历
        region = input[i:i+kernel_h, j:j+kernel_w] # 取出被核滑动到的区域
        output[int(i/stride),int(j/stride)] = torch.sum(region * kernel) # 点乘 并赋值给输出位置的元素
```

做点乘，并赋值给输出位置的元素，这一步就是把输入的区域拿出来，然后跟 kernel 进行一个 点乘  
这样输出就计算完了，代码一写 输出就已经出来了，然后我们就可以返回了，返回output 就好了  
[step1的全部代码](#)

```
input = torch.randn(5,5) # 卷积 输入特征图
kernel = torch.randn(3,3) # 卷积核
# step1 用原始的矩阵运算来实现二维卷积，先不考虑 batch size维度 和 channel维度
def matrix_multiplication_for_conv2d(input,kernel,stride=1):
    input_h,input_w = input.shape
    kernel_h,kernel_w = kernel.shape

    output_h = (floor((input_h - kernel_h)/stride) + 1) # 卷积输出的高度
    output_w = (floor((input_w - kernel_w)/stride) + 1) # 卷积输出的宽度
    output = torch.zeros(output_h,output_w) # 初始化 输出矩阵

    for i in range(0,input_h - kernel_h + 1,stride): # 对高度进行遍历
        for j in range(0,input_w - kernel_w +1,stride): # 对宽度维进行遍历
            region = input[i:i+kernel_h, j:j+kernel_w] # 取出被核滑动到的区域
            output[int(i/stride),int(j/stride)] = torch.sum(region * kernel) # 点乘 并赋值给输出位置的元素
```

现在来思考 这个代码有没有什么问题。首先 回顾一下，

- 这个代码是要计算 input和kernel的二维卷积操作，我们设置了一个stride参数，padding还没用
- 首先我们得到 input形状 和 kernel的形状，然后计算输出的宽度和高度，这个是有公式的；
- 输出特征图的 高度等于 输入特征图的高度 减去 卷积核的高度 除以 stride 并进行 向下 取整 再 +1
- 输出特征图的 宽度 同理，输入特征图的宽度 减去 卷积核的宽度 除以 stride 并进行 向下取整 再 +1

- 有了输出特征图的高度和宽度 我们随机初始化一个 输出矩阵
- 然后呢 我们就随机模拟一个 滑动相乘的操作，首先 我们对 高度维 进行一个遍历，再对 宽度维 进行一个遍历；高度维 是从上到下，宽度维 是从左到右
- 那每次呢，我们要取出被核 滑动到的区域，也就是 `input[i:i+kernel_h]`,这个就是 input取出的 行数的范围，列数的范围 是 `j:j+kernel_w`,这样 就 取到  $3 \times 3$ 的输入区域了 `region = input[i:i+kernel_h, j:j+kernel_w]`
- 并把 这个 区域 作为 element-wise的一个相乘，这个不是矩阵相乘 而是一个 逐元素的相乘，然后再求和 就 得到一个 点乘的结果，再赋值给 output
- 这里赋值的时候需要注意，我们要映射到 对应的 output的一个索引，就是把 i 除以 stride 跟 j 除以 stride来 作为 行索引和 列索引，然后就返回了 output

[逐行 敲了 你自己能复现出来嘛，你也要会逐字复述]

我们可以来 调用一下 传入 input和kernel:

```
#step1 用原始的矩阵运算来实现二维卷积，先不考虑batchsize维度和channel维度
def matrix_multiplication_for_conv2d(input, kernel, stride=1, padding=0):
    input_h, input_w = input.shape
    kernel_h, kernel_w = kernel.shape

    output_h = (floor((input_h - kernel_h)/stride) + 1) # 卷积输出的高度
    output_w = (floor((input_w - kernel_w)/stride) + 1) # 卷积输出的宽度
    output = torch.zeros(output_h, output_w) # 初始化输出矩阵

    for i in range(0, input_h-kernel_h+1, stride): # 对高度维进行遍历
        for j in range(0, input_w-kernel_w+1, stride): # 对宽度维进行遍历
            region = input[i:i+kernel_h, j:j+kernel_w] # 取出被核滑动到的区域
            output[int(i/stride), int(j/stride)] = torch.sum(region * kernel) #点乘，并赋值给输出位置的元素

    return output

mat_mul_conv_output = matrix_multiplication_for_conv2d(input, kernel)

-----
NameError                                 Traceback (most recent call last)
/var/folders/jm/0msk3gg922j555fq8185txch0000gp/T/ipykernel_40477/1975252329.py in <module>
      39     return output
      40
--> 41 mat_mul_conv_output = matrix_multiplication_for_conv2d(input, kernel)
      42

/var/folders/jm/0msk3gg922j555fq8185txch0000gp/T/ipykernel_40477/1975252329.py in matrix_multiplication_for_conv2d(i
put, kernel, stride, padding)
      28     kernel_h, kernel_w = kernel.shape
      29
--> 30     output_h = (floor((input_h - kernel_h)/stride) + 1) # 卷积输出的高度
      31     output_w = (floor((input_w - kernel_w)/stride) + 1) # 卷积输出的宽度
      32     output = torch.zeros(output_h, output_w) # 初始化输出矩阵

NameError: name 'floor' is not defined
```

运行报错，因为 没有导入 math `math.floor()`

然后我们 查看结果

这部分的全部可运行代码：

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

input = torch.randn(5,5) # 卷积 输入特征图
```

```

kernel = torch.randn(3,3) # 卷积核
# step1 用原始的矩阵运算来实现二维卷积，先不考虑 batch size维度 和 channel维度
def matrix_multiplication_for_conv2d(input,kernel,stride=1):
    input_h,input_w = input.shape
    kernel_h,kernel_w = kernel.shape

    output_h = (math.floor((input_h - kernel_h)/stride) + 1) # 卷积输出的高度
    output_w = (math.floor((input_w - kernel_w)/stride) + 1) # 卷积输出的宽度
    output = torch.zeros(output_h,output_w) # 初始化 输出矩阵

    for i in range(0,input_h - kernel_h + 1,stride): # 对高度进行遍历
        for j in range(0,input_w - kernel_w +1,stride): # 对宽度维进行遍历
            region = input[i:i+kernel_h, j:j+kernel_w] # 取出被核滑动到的区域
            output[int(i/stride),int(j/stride)] = torch.sum(region * kernel) # 点乘 并赋值给输出位置的元素

    return output

mat_mul_conv_output = matrix_multiplication_for_conv2d(input,kernel,stride=1)
print(mat_mul_conv_output)

```

结果解读: 在 $5 \times 5$ 的输入 区域上 做一个  $3 \times 3$  的卷积核的卷积 最终 得到一个  $3 \times 3$ 的卷积

现在 我们来验证 自己写的卷积函数 结果是不是 正确，我们把 input 和 kernel 代入到 pytorch中，这个时候 我们要调用 什么api呢？就是F.conv2d这个函数

- 矩阵运算实现卷积的结果
- 调用pytorch api卷积的结果

F.conv2d(), 第一个位置是input、第二个位置是 kernel，但是 这里 我们需要 把 input 扩维一下，因为 pytorch 默认 需要 4维的，所以 我们需要扩维，或者直接 reshape

input.reshape(),形状 就是 1 1 input.shape[0] input.shape[1] 扩成4维，batch size和channel都是1

同样 weight也是一样的 进行替换就好了， bias不用写 我们默认 bias=0

```

F.conv2d(input.reshape((1,1,input.shape[0],input.shape[1])),
         kernel.reshape((1,1,kernel.shape[0],kernel.shape[1])))

```

并传给pytorch\_api\_conv\_output，并打印 验证 pytorch api的结果 和 矩阵运算的结果

```

# 调用PyTorch API卷积的结果
pytorch_api_conv_output = F.conv2d(input.reshape((1,1,input.shape[0],input.shape[1])), \
                                    kernel.reshape((1,1,kernel.shape[0],kernel.shape[1])))

```

同时 因为 pytorch\_api\_conv\_output 是4维 的， 我们再reshape回来，或者 用squeeze(0).squeeze(0)

这里的理解 是 把 每个 0维 挤掉，这样就变成 2维的

```
# 调用PyTorch API卷积的结果
pytorch_api_conv_output = F.conv2d(input.reshape((1,1,input.shape[0],input.shape[1])), \
kernel.reshape((1,1,kernel.shape[0],kernel.shape[1])))
print(pytorch_api_conv_output.squeeze(0).squeeze(0))
```

全部代码：

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

input = torch.randn(5,5) # 卷积 输入特征图
kernel = torch.randn(3,3) # 卷积核
# step1 用原始的矩阵运算来实现二维卷积，先不考虑 batch size维度 和 channel维度
def matrix_multiplication_for_conv2d(input,kernel,stride=1):
    input_h,input_w = input.shape
    kernel_h,kernel_w = kernel.shape

    output_h = (math.floor((input_h - kernel_h)/stride) + 1) # 卷积输出的高度
    output_w = (math.floor((input_w - kernel_w)/stride) + 1) # 卷积输出的宽度
    output = torch.zeros(output_h,output_w) # 初始化 输出矩阵

    for i in range(0,input_h - kernel_h + 1,stride): # 对高度进行遍历
        for j in range(0,input_w - kernel_w +1,stride): # 对宽度维进行遍历
            region = input[i:i+kernel_h, j:j+kernel_w] # 取出被核滑动到的区域
            output[int(i/stride),int(j/stride)] = torch.sum(region * kernel) # 点乘 并赋值给输出位置的元素

    return output

# 矩阵运算实现卷积的结果
mat_mul_conv_output = matrix_multiplication_for_conv2d(input,kernel,stride=1)
print(mat_mul_conv_output)

# 调用pytorch api卷积的结果

pytorch_api_conv_output = F.conv2d(input.reshape((1,1,input.shape[0],input.shape[1])), \
kernel.reshape((1,1,kernel.shape[0],kernel.shape[1])))

print(pytorch_api_conv_output.squeeze(0).squeeze(0))
```

以上实现了 矩阵 运算的实现 和 pytorch api的实现 结果相同

接下来 我们把 padding加上，

改代码 pad要放在一开始的地方

如果padding>0,就对 input做一个pad操作， pad操作用的是F.pad函数，传入要pad的对象 input

然后对input需要上下左右都 pad， 所以第二个参数是长度为 4 的元组，分别是 (padding,padding,padding,padding)参数函数前面是左右、后面是上下

这样得到了新的input，后面代码不需要改了，这样我们实现了 pad操作，(这个pad的操作确实是没想到的)

```
#step1 用原始的矩阵运算来实现二维卷积，先不考虑batchsize维度和channel维度
def matrix_multiplication_for_conv2d(input, kernel, stride=1, padding=0):
    if padding > 0:
        input = F.pad(input, (padding, padding, padding, padding)) ←

    input_h, input_w = input.shape
    kernel_h, kernel_w = kernel.shape

    output_h = (math.floor((input_h - kernel_h)/stride) + 1) # 卷积输出的高度
    output_w = (math.floor((input_w - kernel_w)/stride) + 1) # 卷积输出的宽度
    output = torch.zeros(output_h, output_w) # 初始化输出矩阵

    for i in range(0, input_h-kernel_h+1, stride): # 对高度维进行遍历
        for j in range(0, input_w-kernel_w+1, stride): # 对宽度维进行遍历
            region = input[i:i+kernel_h, j:j+kernel_w] # 取出被核滑动到的区域
            output[int(i/stride), int(j/stride)] = torch.sum(region * kernel) #点乘，并赋值给输出位置的元素

    return output

# 矩阵运算实现卷积的结果
mat_mul_conv_output = matrix_multiplication_for_conv2d(input, kernel)
print(mat_mul_conv_output)

# 调用PyTorch API卷积的结果
pytorch_api_conv_output = F.conv2d(input.reshape((1,1,input.shape[0],input.shape[1])), \
                                    kernel.reshape((1,1,kernel.shape[0],kernel.shape[1])))
print(pytorch_api_conv_output.squeeze(0).squeeze(0))
```

如果我们再加一个 bias, bias默认是 None , 就直接写成 0, 因为我们这里假设通道为1，所以bias可以是个标量

bias在哪里加呢，我们直接在输出的位置加

```
#step1 用原始的矩阵运算来实现二维卷积，先不考虑batchsize维度和channel维度
def matrix_multiplication_for_conv2d(input, kernel, bias=0, stride=1, padding=0):
    if padding > 0:
        input = F.pad(input, (padding, padding, padding, padding)) ←

    input_h, input_w = input.shape
    kernel_h, kernel_w = kernel.shape

    output_h = (math.floor((input_h - kernel_h)/stride) + 1) # 卷积输出的高度
    output_w = (math.floor((input_w - kernel_w)/stride) + 1) # 卷积输出的宽度
    output = torch.zeros(output_h, output_w) # 初始化输出矩阵 ←

    for i in range(0, input_h-kernel_h+1, stride): # 对高度维进行遍历
        for j in range(0, input_w-kernel_w+1, stride): # 对宽度维进行遍历
            region = input[i:i+kernel_h, j:j+kernel_w] # 取出被核滑动到的区域
            output[int(i/stride), int(j/stride)] = torch.sum(region * kernel) + bias #点乘，并赋值给输出位置的元素 ←

    return output
```

现在我们再来验证一下padding 这个功能 是否有效

```

# 矩阵运算实现卷积的结果
mat_mul_conv_output = matrix_multiplication_for_conv2d(input, kernel, padding=1)
print(mat_mul_conv_output)

# 调用PyTorch API卷积的结果
pytorch_api_conv_output = F.conv2d(input.reshape((1,1,input.shape[0],input.shape[1])), \
                                    kernel.reshape((1,1,kernel.shape[0],kernel.shape[1])))
print(pytorch_api_conv_output.squeeze(0).squeeze(0))

tensor([[-0.7855,  1.5590,  1.2712, -3.0939, -3.2357],
       [ 0.4061,  3.4197, -7.4375,  3.3945, -6.0802],
       [ 1.3277, -2.2549, -5.2863,  2.6310, -3.2992],
       [-2.5221, -1.3436, -4.4738,  0.4180,  2.4292],
       [ 0.7528,  0.1661, -3.3339, -1.4516, -2.9226]])
tensor([[ 3.4197, -7.4375,  3.3945],
       [-2.2549, -5.2863,  2.6310],
       [-1.3436, -4.4738,  0.4180]])

```

当 padding=1 的时候大小变成  $5 \times 5$ , 所以这时候输出的 feature map 跟输入这里的 feature map 是一致的, 在这里 padding=1, 在我们这个例子中相当于 padding=same

接下来看, F.conv2d 这个 padding 怎么用。去看万能的官网, 可以传入一个整数。

#### Parameters

- **input** – input tensor of shape (minibatch, in\_channels,  $iH$ ,  $iW$ )
- **weight** – filters of shape (out\_channels,  $\frac{\text{in\_channels}}{\text{groups}}$ ,  $kH$ ,  $kW$ )
- **bias** – optional bias tensor of shape (out\_channels). Default: None
- **stride** – the stride of the convolving kernel. Can be a single number or a tuple ( $sH, sW$ ). Default: 1
- **padding** –  
implicit paddings on both sides of the input. Can be a string {'valid', 'same'}, single number or a tuple ( $padH, padW$ ). Default: 0 padding='valid' is the same as no padding. padding='same' pads the input so the output has the shape as the input. However, this mode doesn't support any stride values other than 1.

传入 padding=1, 可以看到:

```

# 矩阵运算实现卷积的结果
mat_mul_conv_output = matrix_multiplication_for_conv2d(input, kernel, padding=1)
print(mat_mul_conv_output)

# 调用PyTorch API卷积的结果
pytorch_api_conv_output = F.conv2d(input.reshape((1,1,input.shape[0],input.shape[1])), \
                                    kernel.reshape((1,1,kernel.shape[0],kernel.shape[1])), \
                                    padding=1)
print(pytorch_api_conv_output.squeeze(0).squeeze(0))

tensor([[-0.7569, -3.1295, -0.5320, -0.6831, -1.7823],
       [-0.7718, -6.1081,  0.0663,  0.8962, -0.6285],
       [ 5.1785,  2.7980,  2.2681,  5.2662,  2.7395],
       [ 0.8934, -0.3966, -1.2322, -0.2542,  0.8000],
       [-2.4118,  1.8923, -1.4918, -3.4345, -1.3094]])
tensor([[-0.7569, -3.1295, -0.5320, -0.6831, -1.7823],
       [-0.7718, -6.1081,  0.0663,  0.8962, -0.6285],
       [ 5.1785,  2.7980,  2.2681,  5.2662,  2.7395],
       [ 0.8934, -0.3966, -1.2322, -0.2542,  0.8000],
       [-2.4118,  1.8923, -1.4918, -3.4345, -1.3094]])

```

通过 pytorch 的 api 实现的效果, 和自己实现的效果, 结果是一样的。bias 只需要传入一个 tensor 进来就可以了。

#### Parameters

- **input** – input tensor of shape (minibatch, in\_channels,  $iH, iW$ )
- **weight** – filters of shape (out\_channels,  $\frac{\text{in\_channels}}{\text{groups}}, kH, kW$ )
- **bias** – optional bias tensor of shape (out\_channels). Default: None
- **stride** – the stride of the convolving kernel. Can be a single number or a tuple ( $sH, sW$ ). Default: 1
- **padding** –  
implicit paddings on both sides of the input. Can be a string {'valid', 'same'}, single number or a tuple ( $padH, padW$ ). Default: 0 padding='valid' is the same as no padding. padding='same' pads the input so the output has the shape as the input. However, this mode doesn't support any stride values other than 1.

首先定义一个bias，因为我们默认 channel=1，所以我们

`bias = torch.randn(1)` 默认 通道数为1，然后我们依次把bias传入到矩阵相乘 函数，以及F.conv2d这个函数中

```
# 矩阵运算实现卷积的结果
mat_mul_conv_output = matrix_multiplication_for_conv2d(input, kernel, bias=bias, padding=1)
print(mat_mul_conv_output)

# 调用PyTorch API卷积的结果
pytorch_api_conv_output = F.conv2d(input.reshape((1,1,input.shape[0],input.shape[1])), \
                                    kernel.reshape((1,1,kernel.shape[0],kernel.shape[1])), \
                                    padding=1, \
                                    bias=bias)
print(pytorch_api_conv_output.squeeze(0).squeeze(0))

tensor([[ 5.2804,  2.5414,  1.7604, -2.4540, -4.3298],
        [-2.7595, -5.6106,  3.7058,  2.4536, -3.6021],
        [-3.9455,  0.1108,  7.1749,  0.5821, -3.1386],
        [-4.7691,  5.7661,  9.2288, -0.2033, -0.8995],
        [ 2.2326,  3.8205, -2.1789, -3.1649, -0.8256]])
tensor([[ 5.2804,  2.5414,  1.7604, -2.4540, -4.3298],
        [-2.7595, -5.6106,  3.7058,  2.4536, -3.6021],
        [-3.9455,  0.1108,  7.1749,  0.5821, -3.1386],
        [-4.7691,  5.7661,  9.2288, -0.2033, -0.8995],
        [ 2.2326,  3.8205, -2.1789, -3.1649, -0.8256]])
```

再次打印结果，矩阵相乘的结果 和 pytorch api实现的结果相同

以上实现了 padding、bias、stride

全部代码：

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

input = torch.randn(5,5) # 卷积 输入特征图
kernel = torch.randn(3,3) # 卷积核
bias = torch.randn(1)

# step1 用原始的矩阵运算来实现二维卷积，先不考虑 batch size维度 和 channel维度
def matrix_multiplication_for_conv2d(input,kernel,bias=0,stride=1,padding=0):

    if padding >0:
        input = F.pad(input,(padding,padding,padding,padding))
```

```

input_h,input_w = input.shape
kernel_h,kernel_w = kernel.shape

output_h = (math.floor((input_h - kernel_h)/stride) + 1) # 卷积输出的高度
output_w = (math.floor((input_w - kernel_w)/stride) + 1) # 卷积输出的宽度
output = torch.zeros(output_h,output_w) # 初始化 输出矩阵

for i in range(0,input_h - kernel_h + 1,stride): # 对高度进行遍历
    for j in range(0,input_w - kernel_w +1,stride): # 对宽度维进行遍历
        region = input[i:i+kernel_h, j:j+kernel_w] # 取出被核滑动到的区域
        output[int(i/stride),int(j/stride)] = torch.sum(region * kernel) + bias # 点乘 并
赋值给输出位置的元素

return output

# 矩阵运算实现卷积的结果
mat_mul_conv_output = matrix_multiplication_for_conv2d(input,kernel,bias =
bias,stride=1,padding=1)
print(mat_mul_conv_output)

# 调用pytorch api卷积的结果

pytorch_api_conv_output = F.conv2d(input.reshape((1,1,input.shape[0],input.shape[1])),
kernel.reshape((1,1,kernel.shape[0],kernel.shape[1])),
padding=1,bias=bias)

print(pytorch_api_conv_output.squeeze(0).squeeze(0))

```

```

1 import torch
2 import torch.nn as nn
3 import torch.nn.functional as F
4 import math
5
6 input = torch.randn(5,5) # 卷积 输入特征图
7 kernel = torch.randn(3,3) # 卷积核
8 bias = torch.randn(1)
9
10 # step1 用原始的矩阵运算来实现二维卷积, 先不考虑 batch size维度 和 channel维度
11 def matrix_multiplication_for_conv2d(input,kernel,bias=0,stride=1,padding=0):
12
13     if padding >0:
14         input = F.pad(input,(padding,padding,padding,padding))
15
16
17     input_h,input_w = input.shape
18     kernel_h,kernel_w = kernel.shape
19
20     output_h = (math.floor((input_h - kernel_h)/stride) + 1) # 卷积输出的高度
21     output_w = (math.floor((input_w - kernel_w)/stride) + 1) # 卷积输出的宽度
22     output = torch.zeros(output_h,output_w) # 初始化 输出矩阵

```

```

23     for i in range(0,input_h - kernel_h + 1,stride): # 对高度进行遍历
24         for j in range(0,input_w - kernel_w +1,stride): # 对宽度维进行遍历
25             region = input[i:i+kernel_h, j:j+kernel_w] # 取出被核滑动到的区域
26             output[int(i/stride),int(j/stride)] = torch.sum(region * kernel) + bias # 点乘 并赋值给输出位置的元素
27
28     return output
29
30
31 # 矩阵运算实现卷积的结果
32 mat_mul_conv_output = matrix_multiplication_for_conv2d(input,kernel,bias = bias,stride=1,padding=1)
33 print(mat_mul_conv_output)
34
35 # 调用pytorch api卷积的结果
36
37 pytorch_api_conv_output = F.conv2d(input.reshape((1,1,input.shape[0],input.shape[1])),
38                                     kernel.reshape((1,1,kernel.shape[0],kernel.shape[1])),
39                                     padding=1,bias=bias)
40
41 print(pytorch_api_conv_output.squeeze(0).squeeze(0))

```

本节只实现了通道数为1的情况，通过输入输出都是多通道，那么还需要嵌入循环，最外面是输出通道数，第二层是输入通道数，第三层是高度维，第四层是宽度维；这个时候bias是跟输入通道数是一致的。输入通道数为1，那么bias就是一个一维张量；如果输入通道数为2，那么bias就是一个二维张量，padding是不变的。

以上的改变

- 循环
- bias的维度