

时间线：2024年10月31日 开始

【24、手写并验证向量内积实现PyTorch二维卷积】 [https://www.bilibili.com/video/BV1Qb4y1i7n5/?share\\_source=copy\\_web&vd\\_source=5cbbeaf6fa2338b041c25f100ea6483](https://www.bilibili.com/video/BV1Qb4y1i7n5/?share_source=copy_web&vd_source=5cbbeaf6fa2338b041c25f100ea6483)

- 241101
- 241103结束

---

topic：手写并验证向量内积实现pytorch二维卷积

本节课从另一种角度 实现卷积

3 <sub>0</sub>	3 <sub>1</sub>	2 <sub>2</sub>	1	0
0 <sub>2</sub>	0 <sub>2</sub>	1 <sub>0</sub>	3	1
3 <sub>0</sub>	1 <sub>1</sub>	2 <sub>2</sub>	2	3
2	0	0	2	2
2	0	0	0	1

12.0	12.0	17.0
10.0	17.0	19.0
9.0	6.0	14.0

当成长度为9个行向量 与列向量进行矩阵相乘；

3 <sub>0</sub>	3 <sub>1</sub>	2 <sub>2</sub>	1	0
0 <sub>2</sub>	0 <sub>2</sub>	1 <sub>0</sub>	3	1
3 <sub>0</sub>	1 <sub>1</sub>	2 <sub>2</sub>	2	3
2	0	0	2	2
2	0	0	0	1

12.0	12.0	17.0
10.0	17.0	19.0
9.0	6.0	14.0

3	3 <sub>0</sub>	2 <sub>1</sub>	1 <sub>2</sub>	0
0	0 <sub>2</sub>	1 <sub>2</sub>	3 <sub>0</sub>	1
3	1 <sub>0</sub>	2 <sub>1</sub>	2 <sub>2</sub>	3
2	0	0	2	2
2	0	0	0	1

12.0	12.0	17.0
10.0	17.0	19.0
9.0	6.0	14.0

3	3 <sub>0</sub>	2 <sub>1</sub>	1 <sub>2</sub>	0 <sub>2</sub>
0	0	1 <sub>2</sub>	3 <sub>2</sub>	1 <sub>0</sub>
3	1	2 <sub>0</sub>	2 <sub>1</sub>	3 <sub>2</sub>
2	0	0	2	2
2	0	0	0	1

12.0	12.0	17.0
10.0	17.0	19.0
9.0	6.0	14.0

3	3	2	1	0
0 <sub>0</sub>	0 <sub>1</sub>	1 <sub>2</sub>	3	1
3 <sub>2</sub>	1 <sub>2</sub>	2 <sub>0</sub>	2	3
2 <sub>0</sub>	0 <sub>1</sub>	0 <sub>2</sub>	2	2
2	0	0	0	1

12.0	12.0	17.0
10.0	17.0	19.0
9.0	6.0	14.0

3	3	2	1	+
0	0 <sub>0</sub>	1 <sub>1</sub>	3 <sub>2</sub>	1
3	1 <sub>2</sub>	2 <sub>2</sub>	2 <sub>0</sub>	3
2	0 <sub>0</sub>	0 <sub>1</sub>	2 <sub>2</sub>	2
2	0	0	0	1

12.0	12.0	17.0
10.0	17.0	19.0
9.0	6.0	14.0

3	3	2	1	0
0	0	1 <sub>0</sub>	3 <sub>1</sub>	1 <sub>2</sub>
3	1	2 <sub>2</sub>	2 <sub>1</sub>	3 <sub>0</sub>
2	0	0 <sub>0</sub>	2 <sub>1</sub>	2 <sub>2</sub>
2	0	0	0	1

12.0	12.0	17.0
10.0	17.0	19.0
9.0	6.0	14.0

3	3	2	1	0
0	0	1	3	1
3 <sub>0</sub>	1 <sub>1</sub>	2 <sub>2</sub>	2	3
2 <sub>2</sub>	0 <sub>2</sub>	0 <sub>0</sub>	2	2
2	0	0	0	1

12.0	12.0	17.0
10.0	17.0	19.0
9.0	6.0	14.0

3	3	2	1	0
0	0	1	3	1
3 <sub>1</sub>	1 <sub>0</sub>	2 <sub>1</sub>	2 <sub>2</sub>	3
2	0 <sub>2</sub>	0 <sub>2</sub>	2 <sub>0</sub>	2
2	0	0	0	1

12.0	12.0	17.0
10.0	17.0	19.0
9.0	6.0	14.0

3	3	2	1	0
0	0	1	3	1
3	1	2 <sub>0</sub>	2 <sub>1</sub>	3 <sub>2</sub>
2	0	0 <sub>2</sub>	2 <sub>2</sub>	2 <sub>0</sub>
2	0	0	0	1

12.0	12.0	17.0
10.0	17.0	19.0
9.0	6.0	14.0

行数为9的行矩阵 跟 kernel 列向量 进行 相乘，再把乘的结果 reshape一下 作为output的结果

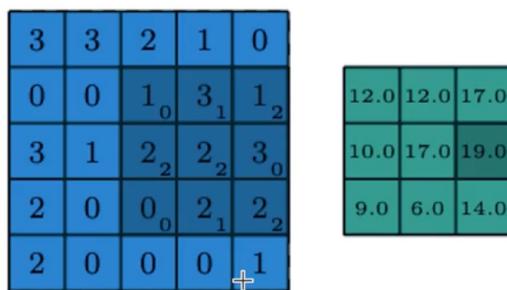
元素相乘 再 求和 就可以看成 矩阵相乘；

第一个矩阵是由9个长度为9的输入矩阵；第二个矩阵由长度为9的列向量的kernel，矩阵相乘 reshape 变成输出的特征图；这是从矩阵拉直的视角看到 矩阵输出。

从另外一种角度，刚刚是 两个 长度为9的 向量 相乘，还有另外一种 内积的方式；

实现一个 长度为25的内积，

具体来说，kernel每次跟输入矩阵  $3 \times 3$ 的范围 进行 矩阵内积；另外一种角度 我们把kernel 填充一下，比如说 第一幅图kernel只有9个数，我们可以理解为kernel有25个数，但是另外的其他数都是0，所以就是说 我们把 每一步的kernel 都进行填充，填充成  $5 \times 5$ 的形状，这样变成了 25的行向量 跟 25的 列向量，进行 矩阵相乘，所以我们把 kernel，每次填充成 input 那么大，用0填充，全部都填充，



比方说 这幅图，kernel 除了深蓝色区域以外，其余位置 全部 填充0，之后再把 kernel 拉成 25的列向量，跟输入25的行向量，进行一个 内积 得到输出

- 输入总是 拉成 行向量，kernel 总是 拉成 列向量，为了 输出 是一个 数（标量）
- 要么跟kernel对齐，要么跟 输入对齐

这种跟 输入对齐的拉直方式 也是一种 处理。这种方式 最终能够 引出转置卷积，其实这种方式得到的kernel，最终会变成一个 (tuo bu li z) 矩阵,之后会讲 而且 不会搞得 那么麻烦，只是 很直观的实现，然后把 那个什么高级 矩阵 转置一下，其实就实现了 transposed convolution,下节课会讲。这是为什么 会讲 $25 \times 25$ 的内积的实现

接下来首先实现 跟kernel对齐的拉直 实现 也就是  $9 \times 9$  的实现；长度为9的输入行向量与长度为9的kernel列向量；

在上次直接的代码的基础上 修改。就是每次 把 region 拉直；

重新 定义一个 函数 叫做 flatten版本

```
# 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，flatten版本
def matrix_multiplication_for_conv2d_flatten(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] # 拉直
            region_vector = torch.flatten(region)
            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
```

接下来讲解 `torch.flatten` 函数，并演示一些常用的api；首先演示 `flatten`, `torch.randn(2,3)` 随机生成  $2 \times 3$  的二阶张量，然后调用 `torch.flatten` 函数，查看会发生什么效果

```
>>> import torch
>>> import torch.nn as nn
>>> import torch.nn.functional as F
>>> # torch.flatten
>>> a=torch.randn(2,3)
>>> a
tensor([[ 1.4483, -0.0792,  0.7185],
        [ 0.6127, -0.5294,  0.3821]])
>>> torch.flatten(a)
tensor([ 1.4483, -0.0792,  0.7185,  0.6127, -0.5294,  0.3821])
```

看到 `a` 被拉直了，`flatten` 的作用 不管 输入 是多少阶的张量，经过 `flatten` 作用以后 都会变成一维的张量，刚好非常适合 卷积的 `flatten` 版本 实现的需求，把 `region` 变成了 `region vector`；

然后考虑 把 `region vector` 放到 `region matrix` 里面去；再把 `region matrix` 跟 `kernel matrix` 进行一个 矩阵相乘；

所以我们在前面要初始化region\_matrix, 那么region\_matrix的维度 应该是什么样的? 行数是output.numel()(这是什么意思?)那列数呢? 列数 是 kernel的numel

```
region_matrix = torch.zeros(output.numel(), kernel.numel())
```

接下来 演示 numel的api, 依然以a这个2行3列的矩阵做演示, 调用一下 a.numel函数

```
>>> a=torch.randn(2,3)
>>> a
tensor([[-0.0617,  0.9972, -0.5227],
       [ 0.7694,  0.0127, -0.3810]])
>>> a.numel()
6
>>> a=torch.randn(2,3,4)
>>> a.numel()
24
```

$$6 = 2 \times 3$$

$$24 = 2 \times 3 \times 4$$

所以 numel返回的是 元素的总数

回到 这里, 我们定义的region matrix, 行数是 output的元素个数, 列数是 kernel的元素个数

region matrix的含义是 存储所有拉直后的 特征区域

接下来 我们把region vector赋值为region matrix的某一行

```
region_matrix = torch.zeros(output.numel(), kernel.numel()) # 存储着所有的拉平后特征区域
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
        region_vector = torch.flatten(region)
        region_matrix[i+j] = region_vector
```

我们把region vector赋值给 region matrix的第  $i+j$  行 【?】

我们是先把 输入的所有特征区域全部 取出来, 存放到region matrix里面去; region matrix 的形状 就是

```
output.numel() * kernel.numel()
```

同时 我们要得到 kernel matrix，因为 最终我们要将 region matrix跟 kernel matrix，进行一个 矩阵的相乘，kernel matrix就比较简单了，就直接把kernel reshape一下 就好了，reshape成一个 列向量

```
kernel_matrix = kernel.reshape((-1, 1))
```

-1行， 1列

现在 我们演示 reshape的api

```
>>> a
tensor([[[ 1.8493, -1.8672,  1.1947,  0.6600],
         [-1.0705,  1.1084, -0.4605, -1.0215],
         [ 0.5336,  1.5858,  1.4711,  0.6741]],

        [[-1.0167, -0.8487,  0.6825, -0.0221],
         [ 0.0667,  1.4666, -0.6753, -0.7569],
         [-1.6863, -2.2171, -0.0179, -0.5898]]])
>>> a.reshape((-1, 1))
tensor([[ 1.8493],
       [-1.8672],
       [ 1.1947],
       [ 0.6600],
       [-1.0705],
       [ 1.1084],
       [-0.4605],
       [-1.0215],
       [ 0.5336],
       [ 1.5858],
       [ 1.4711],
       [ 0.6741],
       [-1.0167],
       [-0.8487],
       [ 0.6825],
       [-0.0221],
       [ 0.0667],
       [ 1.4666],
       [-0.6753],
       [-0.7569],
       [-1.6863],
       [-2.2171],
```

变成一个 列向量；就是把这些元素 按照 列向量的 方式 排列一下

-1表示的是 剩下的 所有维度，不用自己 指定 pytorch 自动去算，1是第二个维度的大小，因为这个a原始的总的numel是24，所以如果 第二个维度是1的话，那么第二个维度就是24，所以最终我们 得到  $24 \times 1$ 的矩阵

所以我们对kernel reshape 得到kernel matrix

```
kernel_matrix = kernel.reshape((-1, 1)) #kernel的列向量形式 或者  
叫 矩阵形式
```

还可以写成：

```
kernel_matrix = kernel.reshape((kernel.numel(), 1))
```

这里的 逻辑是 把 所有kernel的元素 拉成 列向量的形式；所以有几个 kernel 元素 拉成几个 的列向量

```
region_matrix = torch.zeros(output.numel(), kernel.numel()) # 存储着所有的拉平后特征区域
```

region matrix的逻辑是 有几个output的元素 就有几个 行向量；从 input中取出行向量

列就是 kernel的元素个数，因为拉出来的区域 主要逐个元素 与 kernel 相乘

综上：

- region matrix的大小是 output.numel()×kernel.numel()
- kernel matrix的大小是 kernel.numel()×1的大小

所以 可以对 这两个矩阵 进行 矩阵相乘，那就是 region matrix，矩阵相乘 可以用 @ 符号，乘以 kernel matrix，这样得到 output matrix

```
output_matrix = region_matrix @ kernel_matrix
```

，然后再把 output matrix 给 reshape一下

```
output = output_matrix.reshape((output_h, output_w))
```

总的来说 代码是很类似的 但是换了一个思路

原始的

```
# 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

```

flatten版本：

```

# step2 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， flatten版本
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

region_matrix = torch.zeros(output.numel(), kernel.numel()) # 存储着所有拉平后特征区域
kernel_matrix = kernel.reshape(kernel.numel(), 1) # 存储着 kernel 的列向量(矩阵)形式

row_index = 0
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
        region_vector = torch.flatten(region)
        region_matrix[row_index] = region_vector
        row_index += 1

output_matrix = region_matrix @ kernel_matrix
output = output_matrix.reshape((output_h, output_w)) + bias

return output

```

思路是把所有的 input 存到矩阵中，再跟 kernel 进行矩阵相乘；乘出来的结果也是一个列向量，再 reshape一下就好了；

接下来，我们继续进行验证，我们目前都没有考虑 batch 维和 channel 维；

这里用了两个 squeeze() 把四维张量 变成二维的

```

# 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])), \
padding=1, \
bias=bias).squeeze(0).squeeze(0)
torch.allclose()

```

这里我们不打印了，直接用torch.allclose()，比较两个张量所有位置的浮点数是否足够接近；如果都足够接近的话，torch.allclose就会返回一个True的bool量

```
# 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)

# 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).squeeze(0).squeeze(0)
flag = torch.allclose(mat_mul_conv_output, pytorch_api_conv_output)
print(flag)
```

True

这里演示的是上次实现的矩阵运算的卷积和api实现的卷积，结果足够接近，返回true

```
# 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, stride=2)

# 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, stride=2).squeeze(0).squeeze(0)
flag = torch.allclose(mat_mul_conv_output, pytorch_api_conv_output)
print(flag)
```

True

上次没有 stride 这次加上 stride 验证，结果依然为true；接下来我们验证 flatten 的版本

命名为 flatten input 版本，还有 flatten kernel 版本

```
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

# step2 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，flatten版本
def
matrix_multiplication_for_conv2d_flatten(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

    region_matrix = torch.zeros(output.numel(),kernel.numel()) #  

存储着所有拉平后特征区域  

    kernel_matrix = kernel.reshape(kernel.numel(),1) # 存储着  

kernel的 列向量 (矩阵) 形式  

    row_index = 0

    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  

            region_vector = torch.flatten(region)  

            region_matrix[row_index] = region_vector  

            row_index +=1

    output_matrix = region_matrix @ kernel_matrix
    output = output_matrix.reshape((output_h,output_w))+bias

    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=2,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,stride=2).squeeze(0).squeeze(0)

```

```
# 矩阵运算实现卷积的结果 flatten input版本
mat_mul_conv_output_flatten =
matrix_multiplication_for_conv2d_flatten(input,kernel,bias =
bias,stride=2,padding=1)

flag =
torch.allclose(mat_mul_conv_output_flatten,pytorch_api_conv_out
put)
print(flag)
```

总结：

Step1 实现的是 很原始的kernel在input上进行滑动，每两个区域 进行逐元素的相乘，再求和 最后赋值 给output

今天是实现的对 input进行flatten的版本；就是把input对应的region flatten成 行向量 然后和kernel进行相乘。最后再把 乘的结果 reshape再相加

下次 会将 kernel flatten 的版本，再把 权重 进行 转置 实现 转置卷积

为什么叫转置卷积 就是把 weight 进行 转置就可以了；但是需要注意的是 转置卷积 并不是 卷积的 逆过程，只是 形状上 保持一致，从input到output是进行卷积。再把output的形状 卷积成 input的形状 叫做 转置卷积；并不是说 从 output的值 恢复成 input的值，只是说 形状上 进行一个上采样的过程。

接下来 增加 batch size维度和channels维度 实现 卷积，相当于是一个 完整的卷积实现

在step1的基础上实现， Step2 可以自己去加上 batch size 维度 和channels维度，这里 不再演示

思考 怎么在当前的基础山 引入 batch size维度和channels维度，增加怎么样的循环 引入 batch size维度和channels维度，最终实现完整的卷积

重命名函数名称

```
def matrix_multiplication_for_conv2d_full(input, kernel, bias=0, stride=1, padding=0):
```

这里我们的 input 和 kernel 都是四维张量，按照 pytorch 写法，这里要写 8 个数

```
# input, kernel 都是 4 维的张量
if padding > 0:
    input = F.pad(input, (padding, padding, padding, padding))
input_h, input_w = input.shape
```

可以理解为 每个维度 都有上下 (?)

而 pytorch F.pad 这个函数 是从里到外的，这是有点反直觉的，因为 input shape 就是 batch size × input channel × input height × input width，这样的四个维，但是这个 F.pad 是从 里到外的，也就是

```
if padding > 0:
    input = F.pad(input, (padding, padding, padding, padding))
input_h, input_w = input.shape
width
height
```

前面是 width 维度，后面是 height 维度，后面两个是 channels 维度和 batch size 维度；

```
input = F.pad(input, (padding, padding, padding, padding, 0, 0, 0, 0))
```

我们这个 batch size 维度 和 channel 维度 都是不要填充的，所以传入 0，这是第一步 修改 pad 函数

```
bs, in_channel, input_h, input_w = input.shape
```

第二步 input shape 也需要修改，因为 我们现在是 四维的，第一个维度 我们命名为 bs，第二个维度 命名为 in\_channel，第三个维度 和 第四个维度 命名为 input\_h 和 input\_w 不变的

```
out_channel, in_channel, kernel_h, kernel_w = kernel.shape
```

kernel.shape 在 二维卷积中 说过； kernel 的形状也是一个 四维的张量，分别是 output kernel、input channel、kernel h、kernel w，得到我们 kernel 的 shape

另外既然我们要考虑 channel 这个维度，bias 也需要做变化；

bias就不一定是一个 标量，因为 bias的形状 是跟 output channel是一致的，也就是说 有多少个 输出通道，那么bias的长度就是多少，如果我们传入的bias=None，就说明我们的 bias 是一个 全为0的张量，所以 我们也可以加入一个条件验证，如果传入的bias等于 None 那么 就可以把bias设置成一个 全为0的向量，长度是 output channels,因为 bias加到每一个输出通道上。有几个卷积核，加几个bias

```
        ...
        ...
        ...
        ...
    if bias is None:
        bias = torch.zeros(out_channel)
```

以上，我们把pad和bias，修改完了；接下来再去看，首先卷积输出的 高度和宽度是不变的

```
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
```

因为 高度 和宽度是二维的，只跟 kernel的高度 和 宽度有关，这里是不变的。但是 输出 需要改变，之前写的 输出 是一个二维张量

```
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
```

但是现在 我们需要 考虑 batch size 维度 和 channel 维度，这个 output现在 变成一个四维的张量，那么第一维呢 就是 batch size，第二维呢 就是 out channel,第三维和第四维 就是output\_h 和 output\_w,这样 我们初始化 完了 输出矩阵，截止到目前所有修改完的代码：

```
#step3 用原始的矩阵运算来实现二维卷积，考虑batchsize维度和channel维度
def matrix_multiplication_for_conv2d_full(input, kernel, bias=0, stride=1, padding=0):
    # input,kernel都是4维的张量
    if padding > 0:
        input = F.pad(input, (padding, padding, padding, padding, 0, 0, 0, 0))

    bs, in_channel, input_h, input_w = input.shape
    out_channel, in_channel, kernel_h, kernel_w = kernel.shape
    if bias is None:
        bias = torch.zeros(out_channel)

    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(bs, out_channel, output_h, output_w) # 初始化输出矩阵
```

接下来修改遍历，之前仅仅对高度维和宽度维进行了遍历，现在还有输入通道维度、输出通道维、还有batch size维，今天的实现并不考虑效率只是从原理上实现二维卷积

```
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
```

所以在宽度维和高度维还需要增加一个维度，首先是input channel维，之前说过我们会每一个input通道进行一个各自的卷积，所以首先对input channel维进行遍历，另外我们把每个input维，算完以后还需要合并一下，合并一下作为输出通道的结果，

```
for ic in range(in_channel):
    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
```

所以在上面还要加一个输出通道维的遍历，每一个输出通道都是由所有的输入通道维进行一个求和得到的；对输出通道维的遍历只考虑了一个样本

```
for oc in range(out_channel):
    for ic in range(in_channel):
        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
```

在最上面，我们还需要对batch size维再次进行遍历，batch size的索引用idx表示

```
for ind in range(bs):
    for oc in range(out_channel):
        for ic in range(in_channel):
            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

return output
```

这里一共是5个for循环(我的评价：疯了)

看起来比较麻烦但是原理确实是这样的；从里往外看，首先，里面是每一个二维卷积，需要将所有的输入通道的结果加起来，最终作为一个输出通道的结果，最终再考虑其他的batch size的样本

继续修改代码；看region部分 取得input，之前把input当成二维的，因为现在 input 变成了四维的，那么第一维就是 batch size维：ind，第二维 就应该是 input channel维：ic，就是把当前的矩形区域 取出来 作为 region

```
for ind in range(bs):
    for oc in range(out_channel):
        for ic in range(in_channel):
            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[ind, ic, 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
return output
```

下面的代码 继续改动，首先是output；output之前是二维的，现在同样变成了 四维，所以第一维呢，同样还是batch size维：ind，第二维 是 输出通道就是 oc维；

```
region = input[ind, ic, i:i+kernel_h, j:j+kernel_w]
output[int(ind), int(oc), int(i/stride), int(j/stride)] += torch.sum(region * kernel) + bias #点乘，并赋值
```

然后刚刚说过 输出通道 是由所有的 输入通道的 求和；所以这里 就不是一个 简单的等于号了，而是 $+=$

```
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[ind, ic, i:i+kernel_h, j:j+kernel_w] # 取出被核滑动到的区域
        output[int(ind), int(oc), int(i/stride), int(j/stride)] += torch.sum(region * kernel) + bias #点乘，并赋值
```

就是因为 我们要对 所有输入通道的权重结果进行求和；

那么 输入通道 具体是怎么做的？region 我们已经 拿出来了；

然后kernel 我们也要拿，kernel 一共有 $oc \times ic$ 个，所以我们 调用一下 kernel，就是第oc 通道的，第ic通道

```
output[int(ind), int(oc), int(i/stride), int(j/stride)] += torch.sum(region * kernel[oc, ic]) + bias #点乘，并赋值
```

这就是 跟当前的region，进行 元素相乘 的这部分kernel，因为 kernel是四维的，如果把每个kernel 看成长方形的话，那就有 $out\ channel \times in\ channel$ 个，现在 我们只需要 取出一个kernel，按照oc index和ic index，从kernel里面 取出一个 矩形 就好了。

bias这里需要改 不能这么加的， bias 放到output channel这里，就是对 每个output channel都会加一个相应的bias， 所以bias 应该放到out channel 这里去加

```
for ind in range(bs):
    for oc in range(out_channel):
        for ic in range(in_channel):
            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[ind, ic, i:i+kernel_h, j:j+kernel_w] # 取出被核滑动到的区域
                    output[ind, oc, int(i/stride), int(j/stride)] += torch.sum(region * kernel[oc, ic]) #
            output[ind, oc] += bias[oc]
return output
```

也就是说output batch size index 这个维度， oc 也就是 output channel这个维度， 我们需要  $\text{+=}$  一个bias(这里 真难啊 听不懂 就说).bias是一个向量， 我们需要 取出它的 第oc个元素就好了， 这样就实现了 既考虑batch size维度， 同样又考虑了 channel这个维度的， 用原始的矩阵运算实现了 二维卷积 【到底什么叫 一维卷积】， 但是现在基本上都是 调用 pytorch的api实现,所有代码：

```
# step3 用原始的矩阵运算来实现二维卷积，考虑 batch size维度 和 channel
维度
def
matrix_multiplication_for_conv2d_full(input,kernel,bias=0,strid
e=1,padding=0):

    # input kernel 都是4维张量
    if padding >0:
        input = F.pad(input,
(padding,padding,padding,0,0,0))

    bs,in_channel,input_h,input_w = input.shape
    out_channel,in_channel,kernel_h,kernel_w = kernel.shape

    if bias is None:
        bias = torch.zeros(out_channel)

    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(bs,out_channel,output_h,output_w) # 初始化 输出矩阵
```

```

for ind in range(bs):
    for oc in range(out_channel):
        for ic in range(in_channel):
            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[ind, ic, i:i+kernel_h, j:j+kernel_w]
                    # 取出被核滑动到的区域
                    output[ind, oc, int(i/stride), int(j/stride)] += torch.sum(region * kernel[oc, ic]) # 点乘 并赋值给输出位置的元素
            output[ind, oc] += bias[oc]
return output

```

```

#step3 用原始的矩阵运算来实现二维卷积, 考虑batchsize维度和channel维度
def matrix_multiplication_for_conv2d_full(input, kernel, bias=0, stride=1, padding=0):
    # input, kernel都是4维的张量
    if padding > 0:
        input = F.pad(input, (padding, padding, padding, 0, 0, 0))

    bs, in_channel, input_h, input_w = input.shape
    out_channel, in_channel, kernel_h, kernel_w = kernel.shape
    if bias is None:
        bias = torch.zeros(out_channel)

    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(bs, out_channel, output_h, output_w) # 初始化输出矩阵

    for ind in range(bs):
        for oc in range(out_channel):
            for ic in range(in_channel):
                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[ind, ic, i:i+kernel_h, j:j+kernel_w] # 取出被核滑动到的区域
                        output[ind, oc, int(i/stride), int(j/stride)] += torch.sum(region * kernel[oc, ic]) # 点乘, 并赋值
            output[ind, oc] += bias[oc]

    return output

```

(再次复述) 现在考虑 input 和 kernel 都是四维张量, bias 就是一个向量, 长度跟输出通道有关的, stride 和 padding 不讲了, 之前讲过; 首先我们会对 input 进行一个 pad, 那由于 pytorch pad 的 api 比较逆反, 所以我们要把里面的维度写到外面, 然后把 channel 和 batch size 的 pad 放到里面, 然后得到 input shape 和 kernel shape;

input shape 是四维的, 形状 batch size × in channel × 高度 × 宽度;

kernel 的形状也是四维的, 形状是 输出的 channel 数 × 输入的 channel 数 × kernel 的高度 × kernel 的宽度

然后 我们对 bias 进行初始化，如果bias为None的话，我们把它设成一个 全0的向量；

接下来 要去算 输出的 特征图的大小，其实 就是batch size × outchannel ×output h ×output w

接下来做五层循环，最外层是样本层，第二层是输出通道层，第三层 是输入通道层；第四层是 高度；第五层是宽度（这个up主是真牛）；每次把输入特征图的每个通道 计算一个卷积，然后再统一的加到 输出通道上；随后 也不要忘记 在每个输出通道上都加上相应的bias

接下来 我们调用这个函数 证明结果正确

```
# step3 用原始的矩阵运算来实现二维卷积，考虑 batch size维度 和 channel  
维度  
def  
matrix_multiplication_for_conv2d_full(input,kernel,bias=0,stride=1,padding=0):  
  
    # input kernel 都是4维张量  
    if padding >0:  
        input = F.pad(input,  
(padding,padding,padding,padding,0,0,0,0))  
  
    bs,in_channel,input_h,input_w = input.shape  
    out_channel,in_channel,kernel_h,kernel_w = kernel.shape  
  
    if bias is None:  
        bias = torch.zeros(out_channel)  
  
        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(bs,out_channel,output_h,output_w) # 初始化  
    输出矩阵  
  
    for ind in range(bs):  
        for oc in range(out_channel):  
            for ic in range(in_channel):
```

```

        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[ind,ic,i:i+kernel_h, j:j+kernel_w]
# 取出被核滑动到的区域
                output[ind,oc,int(i/stride),int(j/stride)] +=
torch.sum(region * kernel[oc,ic]) # 点乘 并赋值给输出位置的元素
                output[ind,oc] += bias[oc]
        return output

input = torch.randn(2,2,5,5) # bs*in_channel*in_h*in_w
kernel = torch.randn(3,2,3,3) #
out_channel*in_channel*kernel_h*kernel_w
bias = torch.randn(3)

# 验证matrix_multiplication_for_conv2d_full与官方API结果是否一致
pytorch_api_conv_output =
F.conv2d(input,kernel,bias=bias,padding=1,stride=2)
mm_conv2d_full_output =
matrix_multiplication_for_conv2d_full(input,kernel,bias=bias,pa
dding=1,stride=2)
flag =
torch.allclose(pytorch_api_conv_output,mm_conv2d_full_output)
print("all cloas:",flag)

```

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

```

```

# step2 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，flatten版本
def
matrix_multiplication_for_conv2d_flatten(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

    region_matrix = torch.zeros(output.numel(),kernel.numel()) #  

存储着所有拉平后特征区域  

    kernel_matrix = kernel.reshape(kernel.numel(),1) # 存储着  

kernel的 列向量 (矩阵) 形式  

    row_index = 0

    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  

            region_vector = torch.flatten(region)  

            region_matrix[row_index] = region_vector  

            row_index +=1

    output_matrix = region_matrix @ kernel_matrix
    output = output_matrix.reshape((output_h,output_w))+bias

    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=2,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,stride=2).squeeze(0).squeeze(0)

```

```
# 矩阵运算实现卷积的结果 flatten input版本
mat_mul_conv_output_flatten =
matrix_multiplication_for_conv2d_flatten(input,kernel,bias =
bias,stride=2,padding=1)

flag1 =
torch.allclose(mat_mul_conv_output,pytorch_api_conv_output)
flag2 =
torch.allclose(mat_mul_conv_output_flatten,pytorch_api_conv_out
put)
print(flag1)
print(flag2)
```