批处理数据的卷积层矩阵乘法的代码实现

为了高效利用矩阵乘法，一般会批量进行图像的卷积运算。令批量为B，输入数据就是4D数据[B*H*W*D]。每个3D特征图都转变为二维大矩阵，然后按行堆叠在一起，形成一个大矩阵。

在上述例子中，如果有B=10个输入特征图，则大矩阵X的尺寸为30250*363.

算法的核心是将局部窗口数据拉伸为行向量，然后遍历每个特征图的每一个数据窗口，使这些行向量按行堆叠在一起即可。

例：算法代码实现

第一步：将局部敞口的数据拉伸为行向量， 然后比哪里每个特征图的的每一个数据窗口，使这些行向量按行堆叠在一起

```
import numpy as np
import matplotlib.pyplot as plt

# 先设置算法超参数：卷积核尺寸、步长
filter_size = 3 # 卷积核尺寸
filter_size2 = filter_size * filter_size
stride = 1 # 步长
padding = (filter_size - 1) // 2 # 计算0填充padding，'//'表示整数除法。

 # 设置输入特征图的各个参数
 (batch, in_height, in_width, in_depth) = 8, 32, 48, 16
 in_data = np.random.randn(batch, in_height, in_width, in_depth) # 随机生成4D的输入特征图

 # 计算输出特征图的尺寸
 out_height = (in_height - filter_size + 2*padding) // stride + 1 # 计算特征图高度
 out_width = (in_width - filter_size + 2*padding) // stride + 1 # 计算特征图宽度
 out_size = out_height * out_width # 特征图尺寸

 # 输出结果初始化（先用0填充，确定存储空间）
 matric_data = np.zeros((out_size*batch, filter_size2*in_depth)) # 分配输出大矩阵的存储空间， 行数量为out_size*batch

 # padding初始化（0填充）
 padding_data = np.zeros((batch, in_height + 2*padding, in_width + 2*padding, in_depth)) # 进行0填充
 padding_data[:, padding : -padding, padding : -padding, :] = in_data # 进行0填充
```

```
25
26  # 计算卷积运算以步长stride滑动时，在输出数据体上最大能滑动到的位置
27  height_ef = padding_data.shape[1] - filter_size + 1 # 最大滑动高位置
28  width_ef = padding_data.shape[2] # 最大滑动宽位置
29
30  for i_batch in range(batch): # 遍历每一个输入3D特征图
31   i_batch_size = i_batch*out_size # 计算第i_batch个3D特征图的首个局部窗口数据的行位置
32   for i_h, i_height in zip(range(out_height), range(0, height_ef, stride)): # 遍历每一行
33   i_height_size = i_batch_size + i_h * out_width # 计算第i_h行首个局部窗口数据的行位置
34   for i_w, i_width in zip(range(out_width), range(0, width_ef, stride)): # 遍历每一列
35   # 获取局部窗口数据，并使用ravel方法将其拉伸为1D向量，赋值给对应的行
36   matric_data[i_height_size + i_w, :] = padding_data[i_batch, i_height:i_height + filter_size, i_width: i_width + filter_size, :].ravel()
37  # 经过三次for循环后，此时的matric_data是一个shape为（12288，144）的矩阵
38
```

第二步：卷积核组拉伸为列向量

（实际上不需要事先生成四维卷积核，直接生成二维卷积核矩阵即可）

```
1  out_depth = 32 # 设置超参数out_depth
2  weights = 0.01 * np.random.randn(filter_size2 * in_depth, out_depth)
3  # 生成小的随机数初始化权重矩阵(out_depth列)
4  bias = np.zeros((1, out_depth)) # 偏置初始化为0
```

第三步：矩阵相乘和ReLU非线性激活

```
1  filter_data = np.dot(matric_data, weights) + bias # 线性组合 y=ax+b，广播机制
2  filter_data = np.maximum(0, filter_data) # ReLU激活
```

第四步：将filter_data的每一行数据转变为输出4D特征图对应位置的深度维度的数据。

```
1  out_data = np.zeros((batch, out_height, out_width, out_depth)) # 分配4D特征图的存储空间
2
3  for i_batch in range(batch): # 遍历每个3D特征图
4   i_batch_size = i_batch*out_size # 计算第i_batch个3D特征图的首行位置
5   for i_height in range(out_height): # 遍历每一行
6   i_height_size = i_batch_size + i_height*out_width # 计算第i_height行的首行位置
7   for i_width in range(out_width): # 遍历每一列
8   out_data[i_batch, i_height, i_width,:] = filter_data[i_height_size + i_width, :]
9   # 把filter_data对应的行向量赋值给输出4D特征图对应的深度维度
```

对于Numpy中的array多维数组存储模式，array中元素存储模式是先存储最后维度的数据，然后依次存储前一维度数据，最后存储第一个维度的数据。

对于2D数据data = np.random.randn(h ,w),先存储第二维度的数据（先存储行数据，每行数据由w个元素），一行一行地一次存储。对于4D特征图data = np.random,randn(b, h, w, d)是以第四个维度数据（个数为d）为存储单位，依次存储这d个元素， 如果这d个元素存储了w次，相当于存储完第三维数据。如果这d个元素存储量w*h次，相当于存储完第二维数据。

为了快速读取数据，数据存储地址最好连在一起，所以array数组最好是依次读取最后维度的数据， 不要依次读取其他维度的数据。每次读取行的程序比读取列的数据更快，因为行数据块的存储地址连续，列数据块的地址是间断的。 ‘

4D数据的维度定义为： data = np.random.randn(batch, height, width, depth)。最后维度是深度，第一维度是批量。程序依次读取深度维数据，即最后一个维度的数据，以达到加速的目的。

3.池化层

输出特征图的空间尺寸等于输入特征图的空间尺寸会产生三个缺点：

（1）空间尺寸不变，运算量较大

（2）导致全连接层权重数量巨大，导致过拟合

（3）产生冗余

池化概述

池化可以对输入的每一个特征图独立地降低其空间尺寸，保存深度维度不变。

首先对特征图的每个局部窗口数据进行融合，得到一个输出数据，然后采用大于1的步长扫描特征图。最常见的局部窗口尺寸是2*2。步长是2会除去75%的神经元，步长采用3，会去除88.89%的神经元。

池化操作会去除大量的神经元，所以可以看作是一种提纯操作。最常用的是选取局部窗口的最大值，也有不常用的取平均值的操作。

例：池化层操作示意图

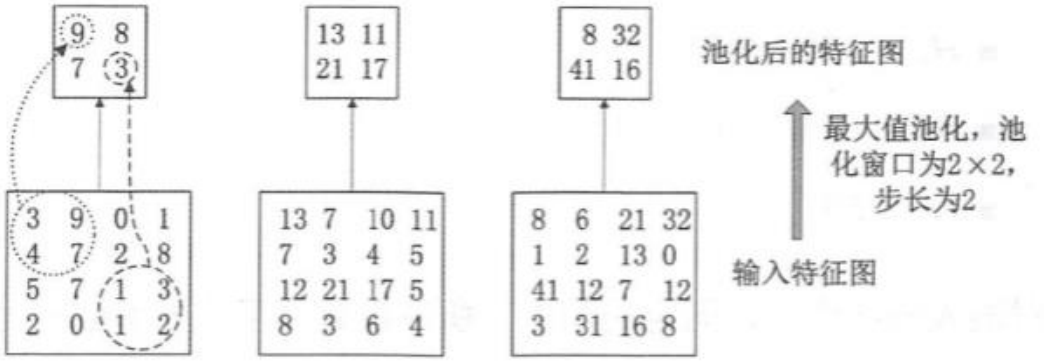

图 4.5　池化层操作示意图

最大值操作可以保持图像的平移不变性，同时适应图像的微小变形和小角度旋转。

池化只减小空间维度尺寸，深度维度的尺寸保持不变。深度维度的尺寸可以看作空间位置出神经园提取的特征数量。随着空间尺寸的减小，神经元感受野越来越大，即神经园观察到的局部区域越来越大，所以需要提取更多的特征，故深度维度一般会随着空间尺寸的减小而增大。

池化层一些参数：

输入特征图尺寸：H1*W1*D1

两个超参数：

滤波器的空间尺寸：F

步长：S

输出特征图的尺寸为：H2*W2*D2，其中：

H2 = （H1 - F）// S + 1

W2 = （W1 - F）// S + 1

D2 = D1

对输入特征图进行固定的操作，所以没有可学习的参数。池化层中很少使用0填充。

池化层代码实现：

池化层将每个局部窗口的数据转化为小矩阵，按行堆叠成大矩阵，然后每行取最大值得到大的列向量，最后转化为3D特征如图。

（1）输入的3D特征图转化为矩阵X：局部区域转化为小矩阵

例如：

输入是 56 × 56 × 96，局部窗口尺寸为 2 × 2，步长为 2 进行池化，取输入中的 2 × 2 × 96 局部数据块，将其转化为尺寸为 96 × 4 的小矩阵，注意是将 96 维的深度向量拉伸为一个列向量，共有 4个深度向量。 以步长为 2 扫描每一个局部窗口 ， 所以输出的宽高均为（56-2）//2+1 =28, 共有 28 × 28 = 784个局部窗口， 784 × 96=75264个行向量，输出矩阵X的尺寸是 75 264 × 4。 因为局部窗口之间没有重叠，所以输入特征图

中的元素在不同的行中没有重复。

（2）最大化池化：提取大矩阵 每行的最大值

matric_data.max(axis=1, keepdims=True)

即得到每个局部窗口的最大值。

例如：

在本例中，这个操作输出的是大列向量[75264*1]

(3)输出新的3D特征图28*28*96，大列向量的每96个元素构成输出3D特征图的一个深度向量。

例：批量数据池化层代码实现：

```
import numpy as np
# 设置超参数
filter_size = 2
filter_size2 = filter_size * filter_size
stride = 2
# 计算输出特征图的维度，分配存储空间
(batch, in_height, in_width, in_depth) = (8, 32, 48, 16)
in_data = np.random.randn(batch, in_height, in_width, in_depth)

 out_height = (in_height - filter_size) // stride + 1
 out_width = (in_width - filter_size) // stride + 1
 out_size = out_width * out_height
 out_depth = in_depth
 out_data = np.zeros((batch, out_height, out_width, out_depth))

 matric_data = np.zeros((out_size*batch*in_depth, filter_size2)) # 分配大矩阵存储空间
 height_ef = in_height - filter_size + 1 # 高度最大值
 width_ef = in_width - filter_size + 1 # 宽度最大值

 for i_batch in range(batch):
  i_batch_size = i_batch * out_size * in_depth # 计算第i_batch个3D特征图的首行位置（要乘以in_depth）
  for i_h, i_height in zip(range(out_height), range(0, height_ef, stride)):
  i_height_size = i_batch_size + i_h * out_width * in_depth
  for i_w, i_width in zip(range(0, out_width*in_depth, in_depth), range(0, width_ef, stride)): # 遍历列
  md = matric_data[i_height_size + i_w:i_height_size + i_w + in_depth, :] # 取出in_depth数量的行向量
```

```
26   src = in_data[i_batch, i_height:i_height + filter_size, i_width:i_width
+ filter_size, :] # 获得输入特征图的局部窗口数据
27   for i in range(filter_size):
28   for j in range(filter_size):
29   md[:, i*filter_size + j] = src[i, j, :] # 赋值空间位置（i，j）的深度维度的
数据
30   # 将空间位置（i，j）位置的深度维度上的in_depth个数据作为一个深度向量，赋值给
大矩阵的列
31
32  matric_data_max_value = matric_data.max(axis=1, keepdims=True) # 最大值滤
波，利用keepdims=True来保持矩阵的维度
33  matric_data_max_pos = matric_data==matric_data_max_value # 保存最大值位
置，以用于计算梯度
34
35  for i_batch in range(batch):
36   i_batch_size = i_batch*out_size*out_depth
37   for i_height in range(out_height):
38   i_height_size = i_batch_size + i_height*out_width*out_depth
39   for i_width in range(out_width):
40   # 赋值深度维度的数据
41   out_data[i_batch, i_height, i_width, :] = matric_data_max_value[i_heigh
t_size + i_width*out_depth:i_height_size + i_width*out_depth + out_depth].r
avel()
```

4.全连接层

全连接层的实现方式有两种：

（1）将输入3D特征图拉伸为1D向量，然后采用常规神经网络的方法进行矩阵乘法。

（2）把全连接层转化为卷积层，常用于物体检测中。

全连接层转化为卷积层

卷积层中的神经元只与输入数据中的一个局部区域连接，并采用参数共享，全连接层中的神经元与输入数据中的全部区域都来凝结，并且参数各不相同。两者可以相互转化。

卷积层转换为全连接层：

任何一个卷积层都能转化为等价的全连接层。此时连接整个输入空间的权重矩阵式一个巨大的矩阵，处理某些特定区域（局部连接）外，其余都是0。不同神经元的矩阵，其非零区域的元素都是相等的（参数共享）。

全连接层转化为卷积层：

全连接层转化为卷积层，卷积层的卷积核尺寸设置为和特征图的空间尺寸一致，不需要0填充，也不需要滑动卷积窗口即可，此时输出空间尺寸为1， 只有一个单独的特征向量，输出变为1*1*depth。例如：一个全赖粘结层，输入特征图为

7*7*512，输出特征图式1*1*1000，这个全连接层可以等效为一个卷积层：F=7， P=0， S=1， K=1000.

全连接层转化为卷积层操作可以在一次前向传播中，让卷积网络在一副更大的输入图像中的不同位置进行卷积 。

例如：

224 × 224 × 3 的输入图像经过多次卷积层和池化层之后得到特征图 7 × 7 × 512，即空间尺寸缩小了32倍，那么 ， 384 × 384 × 3 的大图像经过同样的卷积层和池化层之后，会得到特征图 12 × 12 × 512（ 384/32 = 12 ）。 然后再经过一个全连接层转化得到的卷积层，最终输出为 6 × 6 × 1000 （卷积 F=7，步长 S=I 后空间尺寸为（12-7）// 1＋ 1 = 6 ）。而（384-224)//32 + I = 6，相当于采用尺寸为 F=224 的卷积核，以步长S=32，对 384 x 384 × 3 的图像进行了 6 × 6 =36 次卷积。

全连接层转换为卷积层后的卷积网络只需要进行一次前向传播就和多次卷积的效果一样，相当于共享了计算资源。通常，输入一张尺寸大的图像，使用变换后的卷积网络对空间上很多不同位置的子图像进行评估，得到分支向量，然后求这些分值向量的平均值。

在上例中，得到6*6*1000特征图后，再对每个6*6特征图进行平均，得到最终特征图1*1*1000，效果一般会更好

上述操作只能得到步长为32的卷积效果，如果想用步长小于32，如16，最终会得到11*11*1000的输出，因为（384-224）//16 + 1 = 11。这个问题可以采用两次前向传播解决，第一次在原始图像进行卷积，得到6*6*1000的输出，第二次分别沿高度和宽度平移16个像素，得到新图像368*368*3，然后再新图像上进行卷积，得到5*5*100的输出（（368-224）//32+1=5）。两次结果合并，得到11*11*1000

全连接层代码实现

将超参数固定为S=1，P=0,卷积核尺寸F一般等于输入特征图空间尺寸（将全连接层转化为卷积层），此时输出特征图空间尺寸为1*1，若卷积核尺寸小于输入空间图尺寸，则输出特征图空间尺寸将大于1*1。

当卷积核尺寸F等于输入特征图空间尺寸时，可以采用将输入3D特征图拉伸为1D数据的方式进行计算。

例：全连接成代码实现

```
import numpy as np
import matplotlib.pyplot as plt
last = 0
(batch, in_height, in_width, in_depth) = (8, 32,48,16)
in_data = np.random.randn(batch, in_height, in_width, in_depth)
size = in_height * in_width * in_depth
matric_data = np.zeros((batch, size))
```

```
for i_batch in range(batch):
 matric_data[i_batch] = in_data[i_batch].ravel() # 将3D特征图拉伸为1D向量

out_depth = 32

weights = 0.01 * np.random.randn(size, out_depth)
bias = np.zeros((1, out_depth))

filter_data = np.dot(matric_data, weights) + bias

if not last:
 out_data = np.maximum(0, filter_data) # relu激活
 plt.imshow(out_data)
 plt.show()
```

注：最后一层全连接层输出分值，不需要非线性激活

5.卷积网络的结构

卷积网络的3种基本模块：卷积层（CONV）、池化层（POOL,默认最大值池化）和全连接层（FC）。卷积层和全连接层后面都需要紧接ReLU激活层，而最后一个全连接层后不需要接ReLU激活层。

层的组合模式

卷积网络的最基本结构时：先堆叠一个或多个卷积层进行特征提取，然后接一个池化层进行空间尺度缩小，重复此模式，直到空间尺寸足够小（如7*7或5*5）最后接多个全连接层，其中最后一个全连接层输出类别分值。

例如：

INPUT——>[CONV——>POOL]*3——>FC——>FC

每个卷积层之后紧跟一个池化层，重复了3次。

INPUT→ [CONV→POOL]→[CONVx2 →POOL]×2 →[CONV × 5→POOL]×2 → FC × 2→FC

堆叠卷积层的数量随着网络的加深而变大，如开始的时候，一个卷积层后立即池化，最后是5个卷积层才池化一次。因为随着特征图空间尺寸的减小，神经元的感受野越来越大，提取特征更具有全局性，需要提取更复杂的关系，所以需要更多卷积层。

输入空间尺寸常用是32（CIFAR-10）、64、96（STL-10）、224（ImageNet）、384和512，这些尺寸能被2整除很多次。

卷积层使用小尺寸卷积核（3*3），步长S=1，如果必须使用大的卷积核（5*5或7*7），通常只用在第一个卷积层中，其输入是原始图像，且仅使用一次。

池化层对特征图进行空间降采样，最常用的是2*2感受野的最大值池化，步长为2，另一个不常用的是3*3感受野，步长为2。

实践表明小步长效果更好，步长为1的卷积层不改变输入特征图的空间维度，只对深度维度进行变换。

对卷积层进行零填充不会改变输入特征图的空间尺寸，若不进行零填充，特征图边缘信息就会过快地损失掉，尤其是堆叠很多个卷积层的时候。

堆叠多个卷积层，实现了多个卷积层与非线性激活层的交替结构，比单一的卷积层结构提取的特征更富有表现力，同时可以减小权重的数量。缺点是在进行反向传播时，需要存储中间每个卷积层的激活值，会占用更多内存。

表示学习

多个全连接层可以看作常规神经网络，而卷积网络全连接层之前的多个卷积层和池化层可以看作特征提取器：每层对上一层的输出进行特征变换，把与类别无关的低层变换为与类别密切相关的高层。