**Pytorch实现卷积神经网络四-torch.nn模块详解**

上次推文我们讲述了在torch中有很多核心的模块，本次文章我们来稍微详细的介绍一下torch.nn模块。

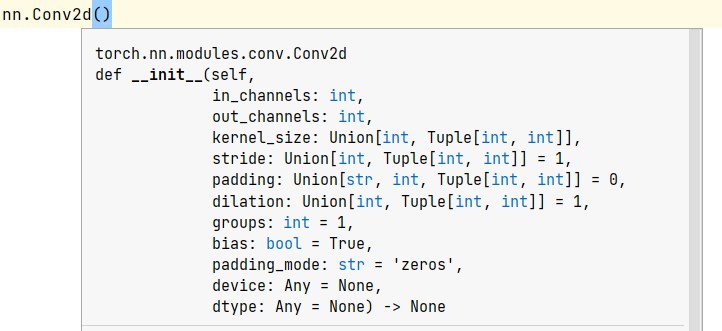
在之前的文章中我们介绍到torch.nn模块中包含了很多的模块，比如说卷积层、池化层、激活层、循环层和全连接层等等，接下来我们就从这几个角度去介绍一下。

1. **torch.nn的卷积**

卷积运算在图像识别、图像分割等应用中十分的常见，卷积作用的加持下稀疏连接、参数共享等方法应运而出，加快了图像领域中深度学习的推进。在torch.nn模块中集成了一维卷积、二维卷积、三维卷积和转置卷积等等，列举如下：

|  |  |
| --- | --- |
| **卷积类** | **功能介绍** |
| [nn.Conv1d](https://pytorch.org/docs/1.12/generated/torch.nn.Conv1d.html" \l "torch.nn.Conv1d" \o "torch.nn.Conv1d)() | 输入信号上应用1D卷积 |
| [nn.Conv2d](https://pytorch.org/docs/1.12/generated/torch.nn.Conv1d.html" \l "torch.nn.Conv1d" \o "torch.nn.Conv1d)() | 输入信号上应用2D卷积 |
| [nn.Conv3d](https://pytorch.org/docs/1.12/generated/torch.nn.Conv1d.html" \l "torch.nn.Conv1d" \o "torch.nn.Conv1d)() | 输入信号上应用3D卷积 |
| nn.ConvTranspose1d() | 输入信号上应用1D转置卷积 |
| nn.ConvTranspose2d() | 输入信号上应用2D转置卷积 |
| nn.ConvTranspose3d() | 输入信号上应用3D转置卷积 |
| [nn.LazyConv1d](https://pytorch.org/docs/1.12/generated/torch.nn.LazyConv1d.html" \l "torch.nn.LazyConv1d" \o "torch.nn.LazyConv1d)() | torch.nn.Conv1d模块，对Conv1d的in\_channels参数进行惰性初始化，该参数是从input.size(1)中推断出来的 |
| [nn.LazyConv2d](https://pytorch.org/docs/1.12/generated/torch.nn.LazyConv2d.html" \l "torch.nn.LazyConv2d" \o "torch.nn.LazyConv2d)() | 意义类似同上 |
| [nn.LazyConv3d](https://pytorch.org/docs/1.12/generated/torch.nn.LazyConv3d.html" \l "torch.nn.LazyConv3d" \o "torch.nn.LazyConv3d)() | 意义类似同上 |
| [nn.LazyConvTranspose1d](https://pytorch.org/docs/1.12/generated/torch.nn.LazyConvTranspose1d.html" \l "torch.nn.LazyConvTranspose1d" \o "torch.nn.LazyConvTranspose1d)() | torch.nn.LazyConvTranspose1d模块，对Conv1d的in\_channels参数进行惰性初始化，该参数是从input.size(1)中推断出来的 |
| [nn.LazyConvTranspose2d](https://pytorch.org/docs/1.12/generated/torch.nn.LazyConvTranspose2d.html" \l "torch.nn.LazyConvTranspose2d" \o "torch.nn.LazyConvTranspose2d)() | 意义类似同上 |
| [nn.LazyConvTranspose3d](https://pytorch.org/docs/1.12/generated/torch.nn.LazyConvTranspose2d.html" \l "torch.nn.LazyConvTranspose2d" \o "torch.nn.LazyConvTranspose2d)() | 意义类似同上 |
| [nn.Unfold](https://pytorch.org/docs/1.12/generated/torch.nn.Unfold.html" \l "torch.nn.Unfold" \o "torch.nn.Unfold)() | 从一个batch图片中，提取出滑动的局部区域块，也就是卷积操作中提取kernel filter对应的滑动窗口 |
| [nn.Fold](https://pytorch.org/docs/1.12/generated/torch.nn.Fold.html" \l "torch.nn.Fold" \o "torch.nn.Fold)() | nn.Unfold()函数的逆操作 |

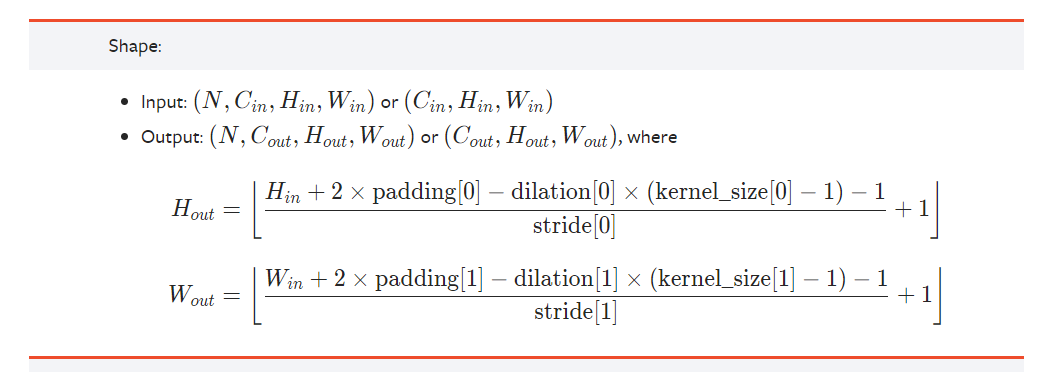
我们来看一下常见的torch.[nn.Conv2d](https://pytorch.org/docs/1.12/generated/torch.nn.Conv1d.html" \l "torch.nn.Conv1d" \o "torch.nn.Conv1d)()方法，这个方法在Torch中的定义如下：



参数含义说明：

1. in\_channels: 输入图像的通道数
2. out\_channels: 输出图像的通道数
3. kernel\_size: 卷积核的大小
4. stride: 卷积的步长，整形或者数组
5. padding: 填充0的数量(层数)
6. dilation:卷积核之间的步幅，即是否使用空洞卷积默认不使用。
7. groups:输入通道到输出通道之间的阻塞连接数
8. bias: 是否增加配置，默认bool = **True**
9. padding\_mode:填充模式，默认 str = **'zeros'**,
10. device=**None**,
11. dtype=**None**

这里就存在一个大家比较关心的问题，输入张量和输出张量之间的映射关系：



**备注：图来自官方文档**

另外大家可以参考文章：

https://mp.weixin.qq.com/s/qVd7lrnouRw3FoNvIl2inw

接下来我们对一张图片进行二维卷积操作，我们编写如下代码：

**import** torch  
**import** numpy **as** np  
**import** torch.nn **as** nn  
**import** matplotlib.pyplot **as** plt  
**from** PIL **import** Image  
*# 读取图片,并转换为Numpy数组*myPic = Image.open(**'cat.jpg'**)  
myPic\_N = np.array(myPic.convert(**'L'**), dtype=np.float32)  
  
plt.figure(figsize=(8, 8))  
plt.imshow(myPic\_N)

plt.axis(**"off"**)  
plt.show()  
print(myPic\_N.dtype)  
print(myPic\_N.shape)

# 输出

float64

(1280, 1920)



上述代码将左边的猫图片转换为一个1280\*1920的灰色图片，然后进行可视化显示。

接下来开始卷积的操作，在此之前我们需要把二维的数组图片转换成四维的数组图片。也就是将[h,w]的图片转换为[batch,channel,h,w]的图片，这样就可以进行卷积的操作，代码如下：

*## 将数组转化为张量*pich,picw = myPic\_N.shape  
mypic\_n\_gray = torch.from\_numpy(myPic\_N.reshape((1,1,pich,picw)))  
print(mypic\_n\_gray.shape)

# 输出如下

torch.Size([1, 1, 1280, 1920])

接下来我们定义一个卷积核，并使用卷积核对图像进行卷积操作：

*# 定义卷积并进行卷积操作*kersize = 7  
*# 数值全部是-1*ker = torch.ones(kersize,kersize,dtype=torch.float32)\*-1  
*# (3,3)的位置为25*ker[3,3] = 25  
ker = ker.reshape((1,1,kersize,kersize))  
*# 设置in\_channels和out\_channels均为1，进行卷积操作*conv2d = nn.Conv2d(in\_channels=1,out\_channels=1,kernel\_size=(kersize,kersize),bias = **False**)  
*# 设置卷积时使用的核*conv2d.weight.data = ker  
*## 对灰度图像进行卷积操作*imconv2dout = conv2d(mypic\_n\_gray)  
print(**'imconv2dout shape:'**,imconv2dout.shape)  
*# ## 对卷积后的输出图像进行复原*imconv2dout\_pic = imconv2dout.data.squeeze()  
print(**'imconv2dout\_pic shape:'**,imconv2dout\_pic.shape)  
print(**"卷积核："**,ker)  
*#可视化卷积后的图像*plt.figure(figsize=(8, 8))  
plt.imshow(imconv2dout\_pic,cmap=plt.cm.gray)  
plt.axis(**"off"**)  
plt.show()

# 打印的输出如下

imconv2dout shape: torch.Size([1, 1, 1274, 1914])

imconv2dout\_pic shape: torch.Size([1274, 1914])

卷积核： tensor([[[[-1., -1., -1., -1., -1., -1., -1.],

[-1., -1., -1., -1., -1., -1., -1.],

[-1., -1., -1., -1., -1., -1., -1.],

[-1., -1., -1., 25., -1., -1., -1.],

[-1., -1., -1., -1., -1., -1., -1.],

[-1., -1., -1., -1., -1., -1., -1.],

[-1., -1., -1., -1., -1., -1., -1.]]]])

经过卷积核运算之后的图像可视化如下所示：



接下来我们设置一下输出的的通道数为2，来进行实验：

*# 核大小*kersize = 7  
ker = torch.ones(kersize,kersize,dtype=torch.float32)\*-1  
ker[3,3] = 25  
ker = ker.reshape((1,1,kersize,kersize))  
print(**"核："**,ker)  
*# 设置输出的通道数为2，进行卷积操作*conv2d = nn.Conv2d(in\_channels=1,out\_channels=2,kernel\_size=(kersize,kersize),bias = **False**)  
*# 设置卷积时使用的核,第一个核使用边缘检测核*print(**'conv2d.weight.data shape:'**,conv2d.weight.data.shape)  
conv2d.weight.data[0] = ker  
*# 对灰度图像进行卷积操作*imconv2dout = conv2d(mypic\_n\_gray)  
print(**'imconv2dout shape:'**,imconv2dout.shape)  
*# 对卷积后的输出图像进行复原*imconv2dout\_pic = imconv2dout.data.squeeze()  
print(**'imconv2dout\_pic shape:'**,imconv2dout\_pic.shape)  
*# 可视化*plt.figure(figsize=(12,6))  
plt.subplot(1,2,1)  
plt.imshow(imconv2dout\_pic[0],cmap=plt.cm.gray)  
plt.axis(**"off"**)  
plt.subplot(1,2,2)  
plt.imshow(imconv2dout\_pic[1],cmap=plt.cm.gray)  
plt.axis(**"off"**)  
plt.show()

# 输出如下

核： tensor([[[[-1., -1., -1., -1., -1., -1., -1.],

[-1., -1., -1., -1., -1., -1., -1.],

[-1., -1., -1., -1., -1., -1., -1.],

[-1., -1., -1., 25., -1., -1., -1.],

[-1., -1., -1., -1., -1., -1., -1.],

[-1., -1., -1., -1., -1., -1., -1.],

[-1., -1., -1., -1., -1., -1., -1.]]]])

conv2d.weight.data shape: torch.Size([2, 1, 7, 7])

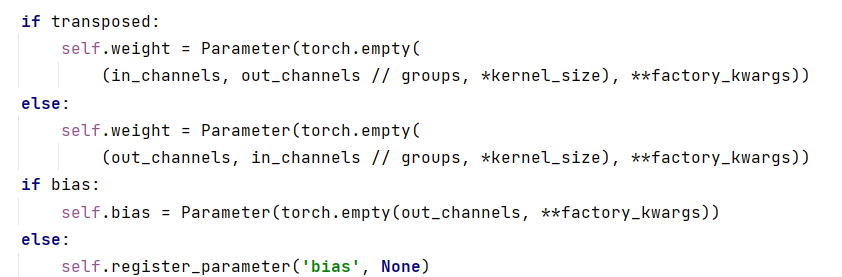
imconv2dout shape: torch.Size([1, 2, 1274, 1914])

imconv2dout\_pic shape: torch.Size([2, 1274, 1914])

图片可视化结果如下：



**在Torch中，当函数torch.nn.Conv2d被调用后会自动初始化weight和bias。**



在上述案例中我们使用：

conv2d.weight.data[0] = ker

自定义了卷积核权值参数，即自定义了卷积核的第一个data[0]权值weight。这也就是为什么我们发现，使用边缘特征提取卷积核很好的提取了图像的边缘特征，而右边图像在随机卷积核（**由torch.nn.Conv2d初始化得到）**的作用下得到的图像会与原始图像很像。大家尝试多次运行词代码，查看每次结果有什么不同之处。

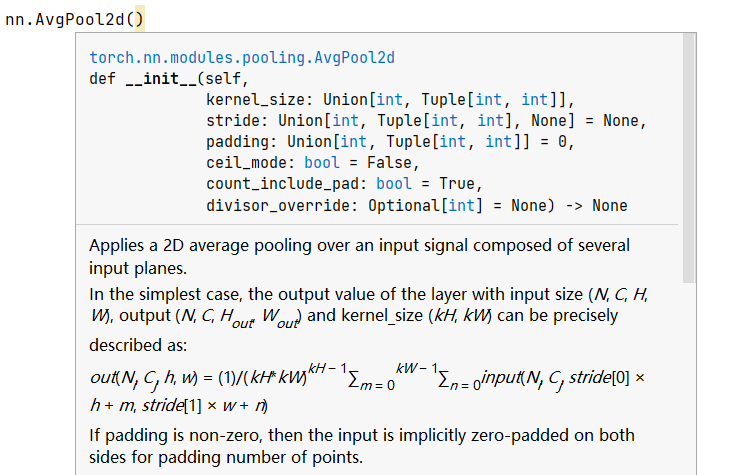
1. **torch.nn的池化**

之前的文章中我们已经说明了池化在操作的作用，主要就是对卷积得到的特征进行进一步的处理，也就是进行降维的操作。从计算资源的角度上来看，池化可以减少特征的维度从而降低计算资源的消耗（内存和CPU）常见的是平均池化和最大值池化。

PyTorch提供了很多供我们使用的池化类，比如说最大池化层MaxPool，最大池化的逆过程MaxUnPool，平均池化层AvgPool等等：

|  |  |
| --- | --- |
| **池化类** | **功能介绍** |
| [nn.MaxPool1d](https://pytorch.org/docs/1.12/generated/torch.nn.MaxPool1d.html" \l "torch.nn.MaxPool1d" \o "torch.nn.MaxPool1d)() | 输入信号上应用1D最大池化 |
| [nn.MaxPool2d](https://pytorch.org/docs/1.12/generated/torch.nn.MaxPool1d.html" \l "torch.nn.MaxPool1d" \o "torch.nn.MaxPool1d)() | 输入信号上应用2D最大池化 |
| [nn.MaxPool3d](https://pytorch.org/docs/1.12/generated/torch.nn.MaxPool1d.html" \l "torch.nn.MaxPool1d" \o "torch.nn.MaxPool1d)() | 输入信号上应用3D最大池化 |
| [nn.MaxUnpool1d](https://pytorch.org/docs/1.12/generated/torch.nn.MaxUnpool1d.html" \l "torch.nn.MaxUnpool1d" \o "torch.nn.MaxUnpool1d)() | 1D最大池化的逆运算 |
| [nn.MaxUnpool2d](https://pytorch.org/docs/1.12/generated/torch.nn.MaxUnpool1d.html" \l "torch.nn.MaxUnpool1d" \o "torch.nn.MaxUnpool1d)() | 2D最大池化的逆运算 |
| [nn.MaxUnpool3d](https://pytorch.org/docs/1.12/generated/torch.nn.MaxUnpool1d.html" \l "torch.nn.MaxUnpool1d" \o "torch.nn.MaxUnpool1d)() | 3D最大池化的逆运算 |
| [nn.AvgPool1d](https://pytorch.org/docs/1.12/generated/torch.nn.AvgPool1d.html" \l "torch.nn.AvgPool1d" \o "torch.nn.AvgPool1d)() | 输入信号上应用1D平均池化 |
| [nn.AvgPool2d](https://pytorch.org/docs/1.12/generated/torch.nn.AvgPool1d.html" \l "torch.nn.AvgPool1d" \o "torch.nn.AvgPool1d)() | 输入信号上应用2D平均纯化 |
| [nn.AvgPool3d](https://pytorch.org/docs/1.12/generated/torch.nn.AvgPool1d.html" \l "torch.nn.AvgPool1d" \o "torch.nn.AvgPool1d)() | 输入信号上应用3D平均池化 |
| [nn.FractionalMaxPool2d](https://pytorch.org/docs/1.12/generated/torch.nn.FractionalMaxPool2d.html" \l "torch.nn.FractionalMaxPool2d" \o "torch.nn.FractionalMaxPool2d)() | 输入信号上应用2D的分数阶平最大池化。  可参考论文：https://arxiv.org/pdf/1412.6071.pdf |
| [nn.FractionalMaxPool3d](https://pytorch.org/docs/1.12/generated/torch.nn.FractionalMaxPool2d.html" \l "torch.nn.FractionalMaxPool2d" \o "torch.nn.FractionalMaxPool2d)() | 输入信号上应用3D的分数阶平最大池化。 |
| [nn.LPPool1d](https://pytorch.org/docs/1.12/generated/torch.nn.LPPool1d.html" \l "torch.nn.LPPool1d" \o "torch.nn.LPPool1d)()/[nn.LPPool2d](https://pytorch.org/docs/1.12/generated/torch.nn.LPPool1d.html" \l "torch.nn.LPPool1d" \o "torch.nn.LPPool1d)() | 输入信号上应用1D/2D的平均功率池化。 |
| [nn.AdaptiveMaxPool1d](https://pytorch.org/docs/1.12/generated/torch.nn.AdaptiveMaxPool1d.html" \l "torch.nn.AdaptiveMaxPool1d" \o "torch.nn.AdaptiveMaxPool1d)() | 输入信号上应用1D的自适应最大池化 |
| [nn.AdaptiveMaxPool2d](https://pytorch.org/docs/1.12/generated/torch.nn.AdaptiveMaxPool1d.html" \l "torch.nn.AdaptiveMaxPool1d" \o "torch.nn.AdaptiveMaxPool1d)() | 输入信号上应用2D的自适应最大池化 |
| [nn.AdaptiveMaxPool3d](https://pytorch.org/docs/1.12/generated/torch.nn.AdaptiveMaxPool1d.html" \l "torch.nn.AdaptiveMaxPool1d" \o "torch.nn.AdaptiveMaxPool1d)() | 输入信号上应用3D的自适应最大池化 |
| [nn.AdaptiveAvgPool1d](https://pytorch.org/docs/1.12/generated/torch.nn.AdaptiveAvgPool1d.html" \l "torch.nn.AdaptiveAvgPool1d" \o "torch.nn.AdaptiveAvgPool1d)() | 输入信号上应用1D的自适应平均池化 |
| [nn.AdaptiveAvgPool2d](https://pytorch.org/docs/1.12/generated/torch.nn.AdaptiveAvgPool1d.html" \l "torch.nn.AdaptiveAvgPool1d" \o "torch.nn.AdaptiveAvgPool1d)() | 输入信号上应用2D的自适应平均池化 |
| [nn.AdaptiveAvgPool3d](https://pytorch.org/docs/1.12/generated/torch.nn.AdaptiveAvgPool1d.html" \l "torch.nn.AdaptiveAvgPool1d" \o "torch.nn.AdaptiveAvgPool1d)() | 输入信号上应用3D的自适应平均池化 |

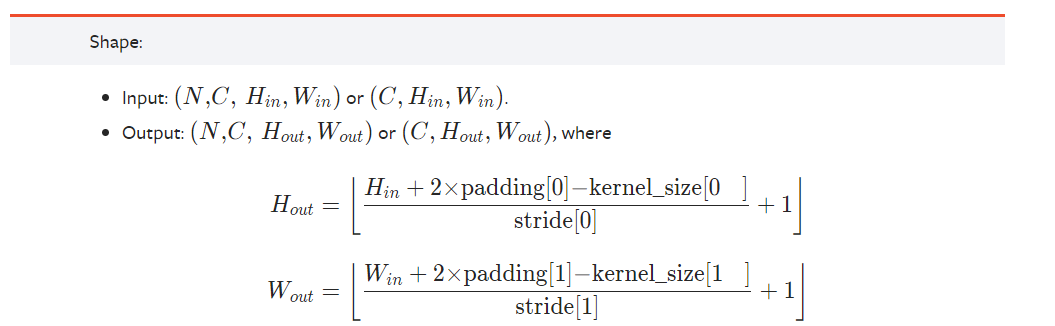
以nn.AvgPool2d为例，该方法的定义如下所示：



参数含义说明：

1. kernel\_size:平均池化的窗口大小
2. stride:卷积的步长，整形或者数组
3. padding:填充0的数量(层数)
4. dilation:卷积核之间的步幅，即是否使用空洞卷积默认不使用。
5. ceil\_mode:True为计算信号大小的时候向上取整，默认False向下取整。
6. count\_include\_pad:默认bool = True将在平均池化计算中包括零填充。
7. divisor\_override:如果指定则池化过程中将池化核内的元素相加再除以divisor\_override，否则将使用池化区域的大小。

一般来说，我们只需要关注kernel\_size、stride和padding这三个参数即可。在池化的过程中我们选用的padding的大小要小于池化核的大小。输入输出的尺寸大小如下所示：



**备注：图来自官方文档**

说到这大家可以关注一下[nn.AdaptiveAvgPool2d](https://pytorch.org/docs/1.12/generated/torch.nn.AdaptiveAvgPool1d.html" \l "torch.nn.AdaptiveAvgPool1d" \o "torch.nn.AdaptiveAvgPool1d)()，因为如果使用这个自适应平均池化，我们在编码的时候只需要关注输出维度的大小output\_size，具体的实现和其他参数的选择由PyTorch自动帮你确定，从这一点上来看还是非常方便的。

<https://zhuanlan.zhihu.com/p/62768077>