PyTorch (/tags/#PyTorch)

PyTorch系列之卷积神经网络 | LeNet

卷积神经网络 | LeNet

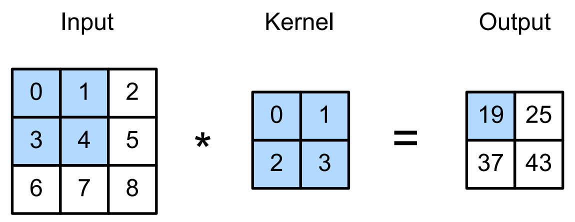
Posted by Young on February 16, 2020

Task05：卷积神经网络基础 | leNet | 卷积神经网络进阶

卷积神经网络基础

二维互相关（cross-correlation）运算

- 二维互相关（cross-correlation）运算的输入是一个二维输入数组和一个二维核（kernel）数组，输出也是一个二维数组，其中核数组通常称为**卷积核或过滤器（filter）**。卷积核的尺寸通常小于输入数组，卷积核在输入数组上滑动，在每个位置上，卷积核与该位置处的输入子数组按元素相乘并求和，得到输出数组中相应位置的元素。



$$0\times0+1\times1+3\times2+4\times3=19$$
$$1\times0+2\times1+4\times2+5\times3=25$$
$$3\times0+4\times1+6\times2+7\times3=37$$
$$4\times0+5\times1+7\times2+8\times3=43$$

```
import torch
import torch.nn as nn

def corr2d(X, K):
        """ >>> X = torch.tensor([[0, 1, 2], [3, 4, 5], [6, 7, 8]]) K = torch.tensor([[0, 1], [2, 3]]) Y = corr2d(X, K) >>> tensor([[19., 25.], [37., 43.]]) """
    H, W = X.shape
    h, w = K.shape
    Y = torch.zeros(H - h + 1, W - w + 1)
    for i in range(Y.shape[0]):
        for j in range(Y.shape[1]):
            Y[i, j] = (X[i: i + h, j: j + w] * K).sum()
    return Y
```

二维卷积层

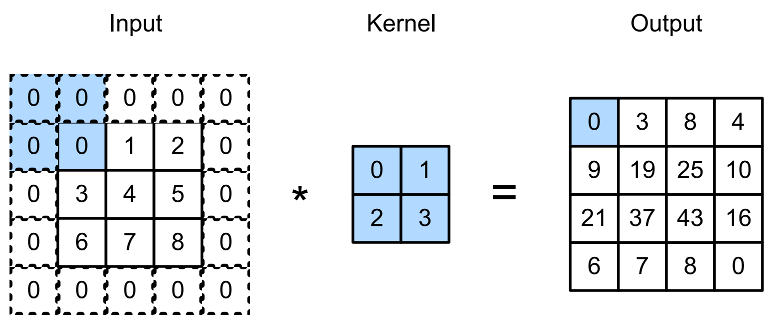
- 二维卷积层将输入和卷积核做互相关运算，并加上一个标量偏置来得到输出。**卷积层的模型参数包括卷积核和标量偏置。**

```python
class Conv2D(nn.Module):
    def __init__(self, kernel_size):
        super(Conv2D, self).__init__()
        self.weight = nn.Parameter(torch.randn(kernel_size))
        self.bias = nn.Parameter(torch.randn(1))

    def forward(self, x):
        return corr2d(x, self.weight) + self.bias
```

- **互相关运算与卷积运算**
    - 卷积层得名于卷积运算，但卷积层中用到的并非卷积运算而是互相关运算。我们**将核数组上下翻转、左右翻转，再与输入数组做互相关运算，这一过程就是卷积运**算。由于卷积层的核数组是可学习的，所以使用互相关运算与使用卷积运算并无本质区别。

padding

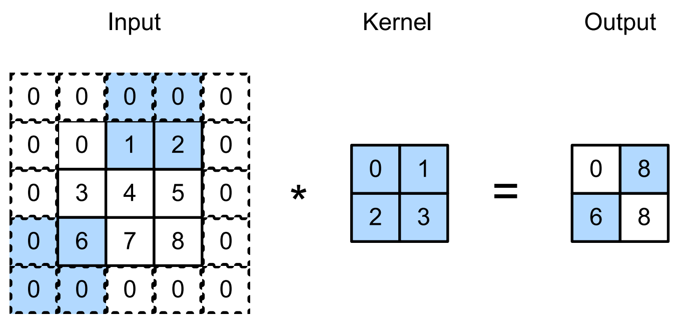
- 填充（padding）是指**在输入高和宽的两侧填充元素（通常是0元素）**



- In general, if we add a total of p_h rows of padding (roughly half on top and half on bottom) and a total of p_w columns of padding (roughly half on the left and half on the right), the output shape will be: $(n_h - k_h + p_h + 1) \times (n_w - k_w + p_w + 1)$.

stride

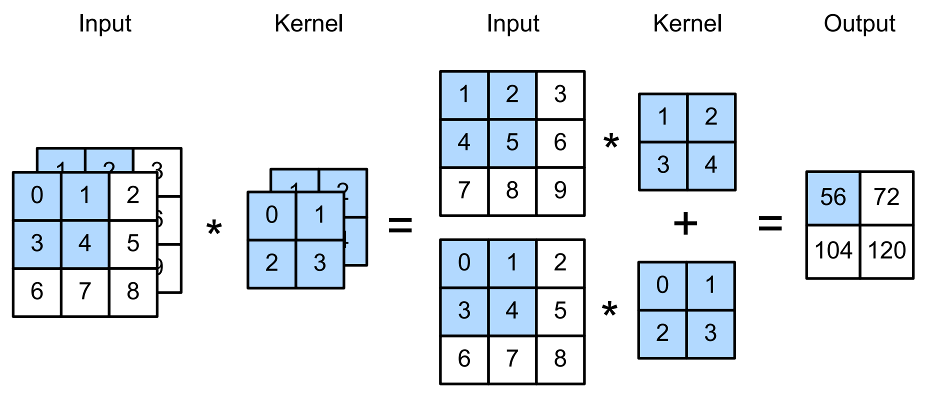
- 在互相关运算中，卷积核在输入数组上滑动，每次滑动的行数与列数即是步幅（stride）



- In general, when the **stride for the height** is s_h and the **stride for the width** is s_w, the output shape is: $\lfloor (n_h - k_h + p_h + s_h) / s_h \rfloor \times \lfloor (n_w - k_w + p_w + s_w) / s_w \rfloor$.
- 当 $p_h = p_w = p$ 时，我们称**填充为 p**；当 $s_h = s_w = s$ 时，我们称**步幅为 s**

多输入通道和多输出通道

- **多输入通道**



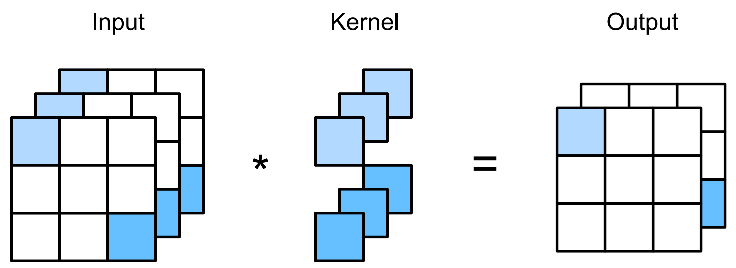
$$(1 \times 1 + 2 \times 2 + 4 \times 3 + 5 \times 4) + (0 \times 0 + 1 \times 1 + 3 \times 2 + 4 \times 3) = 56$$

```
def corr2d_multi_in(X, K):
    """ # First, traverse along the 0th dimension (channel dimension) of X and K. # Then, add them together by using * to turn the result list into a # positional argument of the add_n function """
    return sum(d2l.corr2d(x, k) for x, k in zip(X, K))
```

- **多输出通道**
    - Denote by c_i and c_o the number of input and output channels, respectively, and let k_h and k_w be the height and width of the kernel. To get an output with multiple channels, we can create a **kernel array** of shape $c_i \times k_h \times k_w$ **for each output channel**. We **concatenate** them on the output channel dimension, so that the shape of the convolution kernel is $c_o \times c_i \times k_h \times k_w$.

```
def corr2d_multi_in_out(X, K):
      """ # Traverse along the 0th dimension of K, and each time, perform # cross-correlation operations with input X. All of the results are merged # together using the stack function """
    return np.stack([corr2d_multi_in(X, k) for k in K])
```

1×1 Convolutional Layer



- **Multiple channels** can be used to **extend the model parameters** of the convolutional layer.
- The **1×1** convolutional layer is **equivalent** to the **fully-connected layer**, when applied on a per pixel basis.
- The **1×1** convolutional layer is typically used to **adjust the number of channels** between network layers and to control model **complexity**.

```
def corr2d_multi_in_out_1x1(X, K):
                """ >>> X = np.random.uniform(size=(3, 3, 3)) K = np.random.uniform(size=(2, 3, 1, 1)) Y1 = corr2d_multi_in_out_1x1(X, K) Y2 = corr2d_multi_in_out(X, K) >>> np.abs(Y1 - Y2).sum() < 
    c_i, h, w = X.shape
    c_o = K.shape[0]
    X = X.reshape(c_i, h * w)
    K = K.reshape(c_o, c_i)
    Y = np.dot(K, X)  # Matrix multiplication in the fully connected layer     return Y.reshape(c_o, h, w)
```

卷积层与全连接层的对比

- **二维卷积层经常用于处理图像，与此前的全连接层相比，它主要有两个优势：**
    - 一是全连接层把图像展平成一个向量，在输入图像上相邻的元素可能因为展平操作不再相邻，网络难以捕捉局部信息。而卷积层的设计，天然地具有**提取局部信息的能力。**
    - 二是卷积层的**参数量更少。**不考虑偏置的情况下，一个形状为(ci,co,h,w)的卷积核的参数量是**ci×co×h×w**，与输入图像的宽高无关。假如一个卷积层的输入和输出形状分别是(c1,h1,w1)和(c2,h2,w2)，如果要用全连接层进行连接，参数数量就是**c1×c2×h1×w1×h2×w2**。使用**卷积层可以以较少的参数数量来处理更大的图像。**

`torch.nn.Conv2d`

Parameters

- **in_channels** (*python:int*) – Number of channels in the input image
- **out_channels** (*python:int*) – Number of channels produced by the convolution
- **kernel_size** (*python:int or tuple*) – Size of the convolving kernel
- **stride** (*python:int or tuple, optional*) – Stride of the convolution. Default: 1
- **padding** (*python:int or tuple, optional*) – Zero-padding added to both sides of the input. Default: 0
- **padding_mode** (*string, optional*) – *zeros*
- **dilation** (*python:int or tuple, optional*) – Spacing between kernel elements. Default: 1
- **groups** (*python: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`

Shape:

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

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

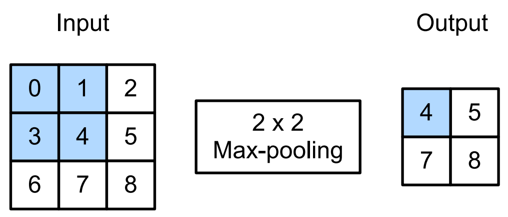
$$W_{out} = \left\lfloor \frac{W_{in} + 2 \times \text{padding}[1] - \text{dilation}[1] \times (\text{kernel\_size}[1] - 1) - 1}{\text{stride}[1]} + 1 \right\rfloor$$

```
X = torch.rand(4, 2, 3, 5)
print(X.shape)

conv2d = nn.Conv2d(in_channels=2, out_channels=3, kernel_size=(3, 5), stride=1, padding=(1, 2))
Y = conv2d(X)
print('Y.shape: ', Y.shape)
print('weight.shape: ', conv2d.weight.shape)
print('bias.shape: ', conv2d.bias.shape)

>>> torch.Size([4, 2, 3, 5])
    Y.shape:  torch.Size([4, 3, 3, 5])
    weight.shape:  torch.Size([3, 2, 3, 5])
    bias.shape:  torch.Size([3])
```

- **kernel 的 in_channels 与 C_{in} 相同, out_channels 与 C_{out} 相同**

Maximum Pooling and Average Pooling**

- Like convolutional layers, pooling operators consist of a fixed-shape window that is **slid over all regions in the input according to its stride**, computing a single output for each location traversed by the fixed-shape window (sometimes known as the *pooling window*). However, unlike the cross-correlation computation of the inputs and kernels in the convolutional layer, **the pooling layer contains no parameters** (there is no *filter*). Instead, pooling operators are deterministic, **typically calculating either the maximum or the average value** of the elements in the pooling window. These operations are called *maximum pooling* (*max pooling* for short) and *average pooling*, respectively.



$$
\begin{aligned}
\max(0,1,3,4) &= 4 \\
\max(1,2,4,5) &= 5 \\
\max(3,4,6,7) &= 7 \\
\max(4,5,7,8) &= 8
\end{aligned}
$$

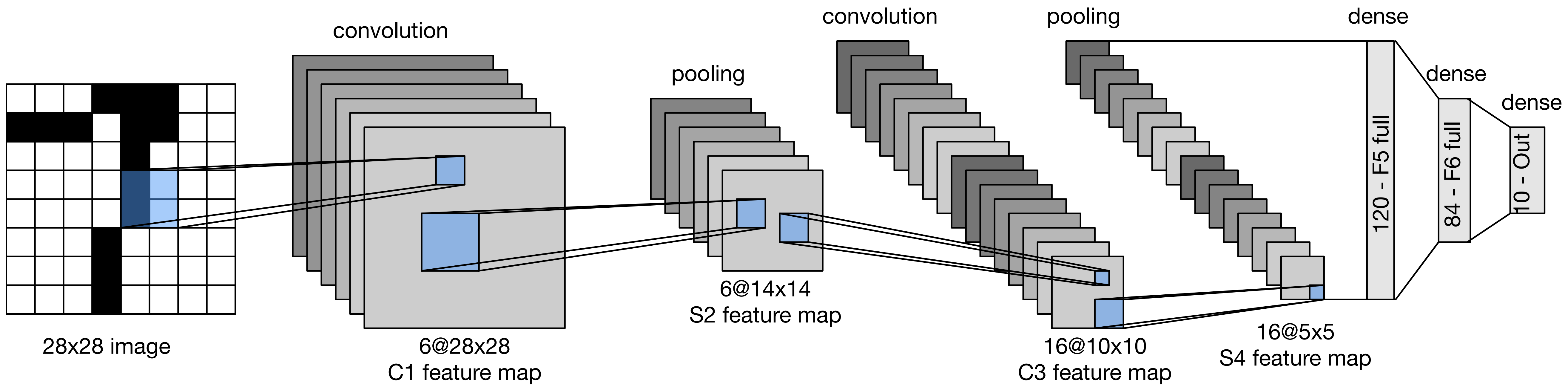
题目

- 假如你用全连接层处理一张256×256的彩色（RGB）图像，输出包含1000个神经元，在使用偏置的情况下，参数数量是：
  - 图像展平后长度为3×256×256，权重参数和偏置参数的数量 3×256×256×1000+1000=196609000
- 假如你用全连接层处理一张256×256的彩色（RGB）图像，卷积核的高宽是3×3，输出包含10个通道，在使用偏置的情况下，这个卷积层共有多少个参数：
  - 输入通道数是3，输出通道数是10，所以参数数量是10×3×3×3+10=280
- `conv2d = nn.Conv2d(in_channels=3, out_channels=4, kernel_size=3, padding=2)`，输入一张形状为3×100×100的图像，输出的形状为：
  - 输出通道数是4，上下两侧总共填充4行，卷积核高度是3，所以输出的高度是104 - 3 + 1=102，宽度同理可得
- 1×1卷积可以看作是通道维上的全连接
- 卷积层通过填充、步幅、输入通道数、输出通道数等调节输出的形状

- 两个连续的3×3卷积核的感受野与一个5×5卷积核的感受野相同
- 池化层没有模型参数（池化层直接对窗口内的元素求最大值或平均值，并没有模型参数参与计算）
- 池化层通常会减小特征图的高和宽
- 池化层的输入和输出具有相同的通道数

LeNet 模型

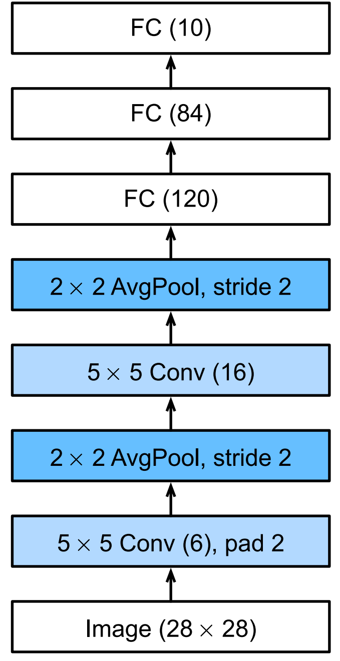
LeNet



- LeNet分为**卷积层块和全连接层块**两个部分。
  - 卷积层块里的基本单位是卷积层后接平均池化层：卷积层用来识别图像里的空间模式，如线条和物体局部，之后的平均池化层则用来降低卷积层对位置的敏感性。

    卷积层块由两个这样的基本单位重复堆叠构成。在卷积层块中，每个卷积层都使用5×5的窗口，并在输出上使用sigmoid激活函数。**第一个卷积层输出通道数为6，第二个卷积层输出通道数则增加到16。**

    全连接层块含3个全连接层。它们的输出个数分别是120、84和10，其中10为输出的类别个数。


FC (10)
FC (84)
FC (120)
2 × 2 AvgPool, stride 2
5 × 5 Conv (16)
2 × 2 AvgPool, stride 2
5 × 5 Conv (6), pad 2
Image (28 × 28)

```
#net
class Flatten(torch.nn.Module):  #展平操作
    def forward(self, x):
        return x.view(x.shape[0], -1)

class Reshape(torch.nn.Module): #将图像大小重定型
    def forward(self, x):
        return x.view(-1,1,28,28)      #(B x C x H x W)

net = torch.nn.Sequential(     #Lelet
    Reshape(),
    nn.Conv2d(in_channels=1, out_channels=6, kernel_size=5, padding=2),
    #b*1*28*28  =>b*6*(28-5+2*2+1)*(28-5+2*2+1) = b*6*28*28
    nn.Sigmoid(),
    nn.AvgPool2d(kernel_size=2, stride=2),
    #b*6*28*28  =>b*6*14*14
    nn.Conv2d(in_channels=6, out_channels=16, kernel_size=5),
    #b*6*14*14  =>b*16*(14-5+1)*(14-5+1) = b*16*10*10
    nn.Sigmoid(),
    nn.AvgPool2d(kernel_size=2, stride=2),
    #b*16*10*10  => b*16*5*5
    Flatten(),
    #b*16*5*5   => b*400
    nn.Linear(in_features=16*5*5, out_features=120),
    nn.Sigmoid(),
    nn.Linear(120, 84),
    nn.Sigmoid(),
    nn.Linear(84, 10)
)

>>> # print
                X = torch.randn(size=(1,1,28,28), dtype = torch.float32)
                for layer in net:
    X = layer(X)
    print(layer.__class__.__name__,'output shape: \t',X.shape)

>>> Reshape output shape:        torch.Size([1, 1, 28, 28])
    Conv2d output shape:         torch.Size([1, 6, 28, 28])
    Sigmoid output shape:        torch.Size([1, 6, 28, 28])
    AvgPool2d output shape:      torch.Size([1, 6, 14, 14])
    Conv2d output shape:         torch.Size([1, 16, 10, 10])
    Sigmoid output shape:        torch.Size([1, 16, 10, 10])
    AvgPool2d output shape:      torch.Size([1, 16, 5, 5])
    Flatten output shape:        torch.Size([1, 400])
    Linear output shape:         torch.Size([1, 120])
    Sigmoid output shape:        torch.Size([1, 120])
    Linear output shape:         torch.Size([1, 84])
    Sigmoid output shape:        torch.Size([1, 84])
    Linear output shape:         torch.Size([1, 10])
```

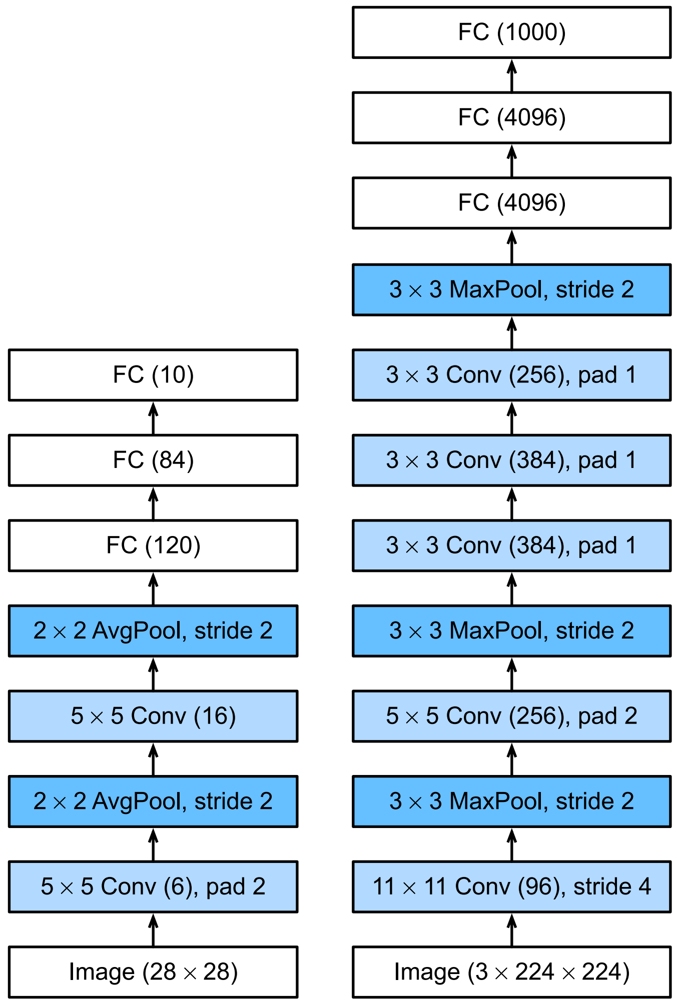
题目

- LeNet主要分为两个部分：卷积层块和全连接层块
- LeNet模型中，90%以上的参数集中在全连接层块
- LeNet在连接卷积层块和全连接层块时，需要做一次展平操作
- LeNet的卷积层块交替使用卷积层和池化层。

- 使用形状为2×2，步幅为2的池化层，会将高和宽都减半
- 卷积神经网络通过使用滑动窗口在输入的不同位置处重复计算，减小参数数量
- 在通过卷积层或池化层后，输出的高和宽可能减小，为了尽可能保留输入的特征，我们可以在减小高宽的同时增加通道数

卷积神经网络进阶

Deep Convolutional Neural Networks（AlexNet）



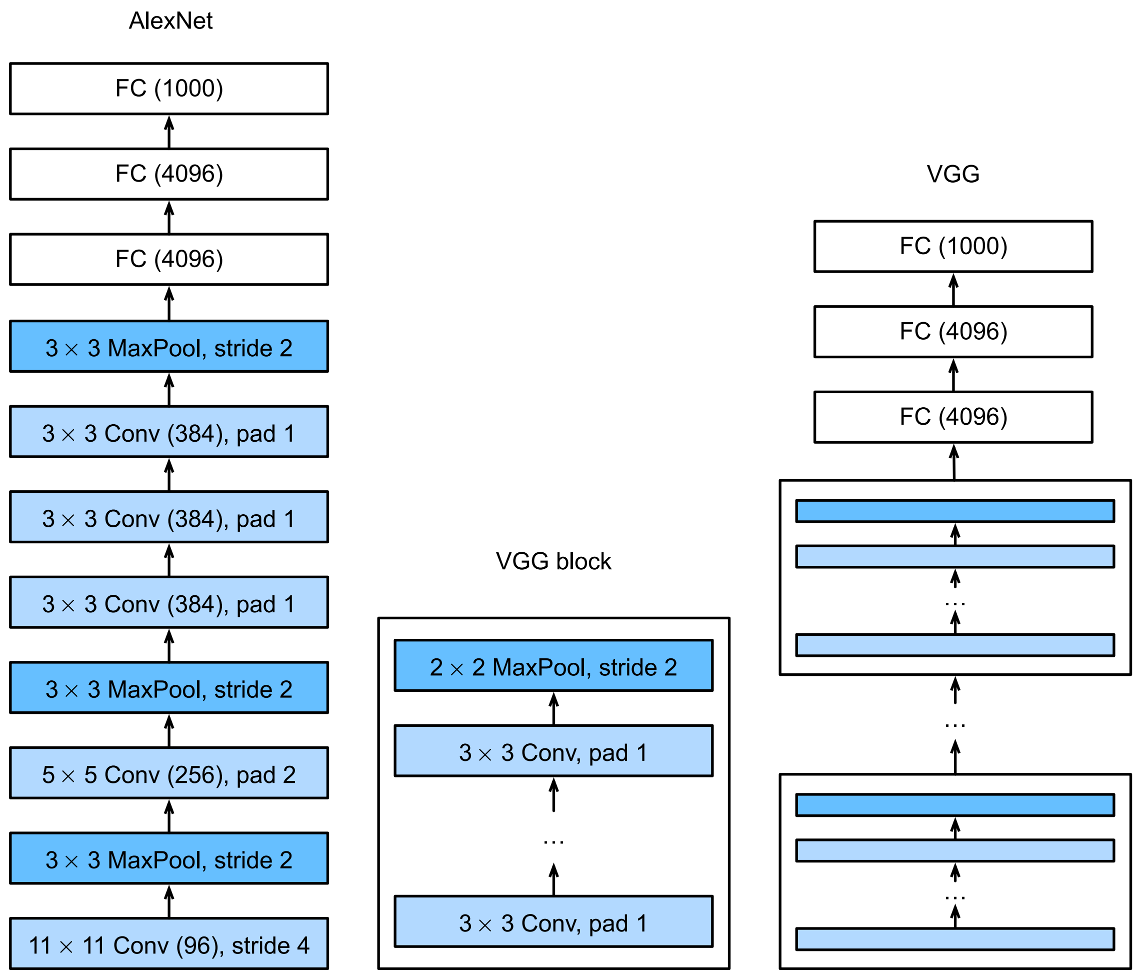
- **LeNet: 在大的真实数据集上的表现并不尽如人意。** 1.**神经网络计算复杂。** 2.**还没有大量深入研究参数初始化和非凸优化算法等诸多领域。**

  机器学习的特征提取: 手工定义的特征提取函数 神经网络的特征提取：通过学习得到数据的多级表征，并逐级表示越来越抽象的概念或模式。

  神经网络发展的限制:数据、硬件
- **AlexNet：首次证明了学习到的特征可以超越手工设计的特征，从而一举打破计算机视觉研究的前状。**
  - **特征：**
  - **8层变换**，其中有5层卷积和2层全连接隐藏层，以及1个全连接输出层。
    - 将sigmoid激活函数改成了更加简单的**ReLU激活函数。**
    - 用**Dropout**来控制全连接层的模型复杂度。
    - **引入数据增强，如翻转、裁剪和颜色变化**，从而进一步扩大数据集来缓解过拟合。

```
class AlexNet(nn.Module):
    def __init__(self):
        super(AlexNet, self).__init__()
        self.conv = nn.Sequential(
            nn.Conv2d(1, 96, 11, 4), # in_channels, out_channels, kernel_size, stride, padding
            nn.ReLU(),
            nn.MaxPool2d(3, 2), # kernel_size, stride
            # 减小卷积窗口，使用填充为2来使得输入与输出的高和宽一致，且增大输出通道数
            nn.Conv2d(96, 256, 5, 1, 2),
            nn.ReLU(),
            nn.MaxPool2d(3, 2),
            # 连续3个卷积层，且使用更小的卷积窗口。除了最后的卷积层外，进一步增大了输出通道数。
            # 前两个卷积层后不使用池化层来减小输入的高和宽
            nn.Conv2d(256, 384, 3, 1, 1),
            nn.ReLU(),
            nn.Conv2d(384, 384, 3, 1, 1),
            nn.ReLU(),
            nn.Conv2d(384, 256, 3, 1, 1),
            nn.ReLU(),
            nn.MaxPool2d(3, 2)
        )
         # 这里全连接层的输出个数比LeNet中的大数倍。使用丢弃层来缓解过拟合
        self.fc = nn.Sequential(
            nn.Linear(256*5*5, 4096),
            nn.ReLU(),
            nn.Dropout(0.5),
            #由于使用CPU镜像，精简网络，若为GPU镜像可添加该层
            #nn.Linear(4096, 4096),
            #nn.ReLU(),
            #nn.Dropout(0.5),

            # 输出层。由于这里使用Fashion-MNIST，所以用类别数为10，而非论文中的1000
            nn.Linear(4096, 10),
        )

    def forward(self, img):

        feature = self.conv(img)
        output = self.fc(feature.view(img.shape[0], -1))
        return output
```

Networks Using Blocks（VGG)



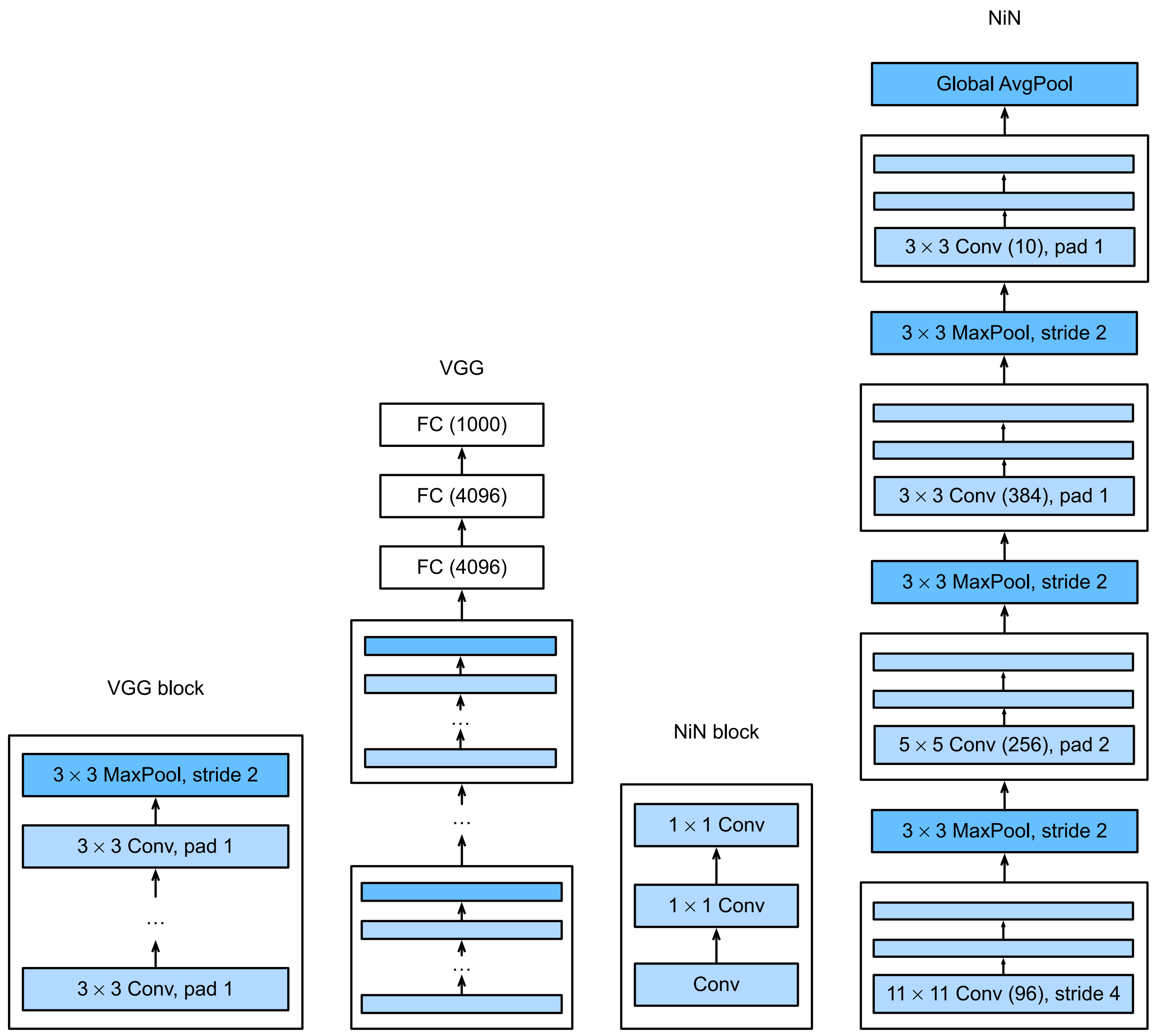
- **VGG：通过重复使用简单的基础块来构建深度模型。** Block: 数个相同的填充为1、窗口形状为3×3的卷积层,接上一个步幅为2、窗口形状为2×2的最大池化层。

  卷积层保持输入的高和宽不变，而池化层则对其减半。

```
def vgg_block(num_convs, in_channels, out_channels): #卷积层个数，输入通道数，输出通道数
    blk = []
    for i in range(num_convs):
        if i == 0:
            blk.append(nn.Conv2d(in_channels, out_channels, kernel_size=3, padding=1))
        else:
            blk.append(nn.Conv2d(out_channels, out_channels, kernel_size=3, padding=1))
        blk.append(nn.ReLU())
    blk.append(nn.MaxPool2d(kernel_size=2, stride=2)) # 这里会使宽高减半
    return nn.Sequential(*blk)
```

```
def vgg(conv_arch, fc_features, fc_hidden_units=4096):
    net = nn.Sequential()
    # 卷积层部分
    for i, (num_convs, in_channels, out_channels) in enumerate(conv_arch):
        # 每经过一个vgg_block都会使宽高减半
        net.add_module("vgg_block_" + str(i+1), vgg_block(num_convs, in_channels, out_channels))
    # 全连接层部分
    net.add_module("fc",
                nn.Sequential(d2l.FlattenLayer(),
        nn.Linear(fc_features, fc_hidden_units),
        nn.ReLU(),
        nn.Dropout(0.5),
        nn.Linear(fc_hidden_units, fc_hidden_units),
        nn.ReLU(),
        nn.Dropout(0.5),
        nn.Linear(fc_hidden_units, 10)
        ))
    return net


>>> conv_arch = ((1, 1, 64), (1, 64, 128), (2, 128, 256), (2, 256, 512), (2, 512, 512))
    # 经过5个vgg_block, 宽高会减半5次, 变成 224/32 = 7
    fc_features = 512 * 7 * 7 # c * w * h
    fc_hidden_units = 4096 # 任意

>>> net = vgg(conv_arch, fc_features, fc_hidden_units)
    X = torch.rand(1, 1, 224, 224)

    # named_children获取一级子模块及其名字(named_modules会返回所有子模块,包括子模块的子模块)
    for name, blk in net.named_children():
        X = blk(X)
        print(name, 'output shape: ', X.shape)

>>> vgg_block_1 output shape:  torch.Size([1, 64, 112, 112])
    vgg_block_2 output shape:  torch.Size([1, 128, 56, 56])
    vgg_block_3 output shape:  torch.Size([1, 256, 28, 28])
    vgg_block_4 output shape:  torch.Size([1, 512, 14, 14])
    vgg_block_5 output shape:  torch.Size([1, 512, 7, 7])
    fc output shape:  torch.Size([1, 10])
```

Network in Network（NiN）



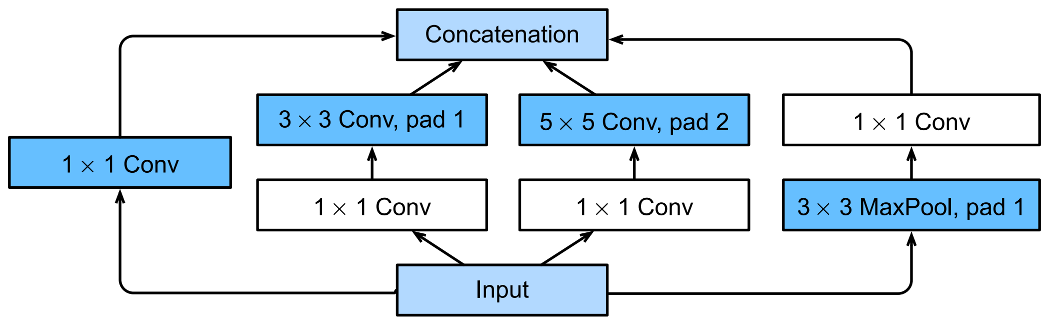
- **LeNet、AlexNet和VGG：先以由卷积层构成的模块充分抽取 空间特征，再以由全连接层构成的模块来输出分类结果。**
- **NiN：串联多个由卷积层和“全连接”层构成的小网络来构建一个深层**网络。 用了输出通道数等于标签类别数的NiN块，然后使用全局平均池化层对每个通道中所有元素求平均并直接用于分类。
    - **1×1卷积核作用**
        - 放缩通道数：**通过控制卷积核的数量达到通道数的放缩。**
        - 增加非线性。**1×1卷积核的卷积过程相当于全连接层的计算过程，并且还加入了非线性激活函数，从而可以增加网络的非线性。**
        - **计算参数少**
    - **NiN重复使**用由卷积层和代替全连接层的1×1卷积层构成的NiN块来构建深层网络。
    - **NiN去除了容易造成过拟合的全连接输出层**，而是将其替换成输出通道数等于标签类别数 的NiN块和全局平均池化层。
    - **NiN的以上设计思想影响了后面一系列卷积神经网络的设计。**

```
def nin_block(in_channels, out_channels, kernel_size, stride, padding):
    blk = nn.Sequential(nn.Conv2d(in_channels, out_channels, kernel_size, stride, padding),
                        nn.ReLU(),
                        nn.Conv2d(out_channels, out_channels, kernel_size=1),
                        nn.ReLU(),
                        nn.Conv2d(out_channels, out_channels, kernel_size=1),
                        nn.ReLU())
    return blk
```

```
class GlobalAvgPool2d(nn.Module):
    # 全局平均池化层可通过将池化窗口形状设置成输入的高和宽实现
    def __init__(self):
        super(GlobalAvgPool2d, self).__init__()
    def forward(self, x):
        return F.avg_pool2d(x, kernel_size=x.size()[2:])

net = nn.Sequential(
    nin_block(1, 96, kernel_size=11, stride=4, padding=0),
    nn.MaxPool2d(kernel_size=3, stride=2),
    nin_block(96, 256, kernel_size=5, stride=1, padding=2),
    nn.MaxPool2d(kernel_size=3, stride=2),
    nin_block(256, 384, kernel_size=3, stride=1, padding=1),
    nn.MaxPool2d(kernel_size=3, stride=2),
    nn.Dropout(0.5),
    # 标签类别数是10
    nin_block(384, 10, kernel_size=3, stride=1, padding=1),
    GlobalAvgPool2d(),
    # 将四维的输出转成二维的输出，其形状为(批量大小, 10)
    d2l.FlattenLayer())
```

```
>>> X = torch.rand(1, 1, 224, 224)
    for name, blk in net.named_children():
        X = blk(X)
        print(name, 'output shape: ', X.shape)

>>> 0 output shape:  torch.Size([1, 96, 54, 54])
    1 output shape:  torch.Size([1, 96, 26, 26])
    2 output shape:  torch.Size([1, 256, 26, 26])
    3 output shape:  torch.Size([1, 256, 12, 12])
    4 output shape:  torch.Size([1, 384, 12, 12])
    5 output shape:  torch.Size([1, 384, 5, 5])
    6 output shape:  torch.Size([1, 384, 5, 5])
    7 output shape:  torch.Size([1, 10, 5, 5])
    8 output shape:  torch.Size([1, 10, 1, 1])
    9 output shape:  torch.Size([1, 10])
```

Networks with Parallel Concatenations（GoogLeNet）



- GoogLeNet
  - **由Inception基础块组成。**
  - Inception块相当于一个**有4条线路的**子网络。它通过不同窗口形状的卷积层和最大池化层来并行抽取信息，并使用1×1卷积层减少通道数从而降低模型复杂度。
  - 可以自**定义的超参数是每个层的输出通道数**，我们以此来控制模型复杂度。

```
class Inception(nn.Module):
    # c1 - c4为每条线路里的层的输出通道数
    def __init__(self, in_c, c1, c2, c3, c4):
        super(Inception, self).__init__()
        # 线路1，单1 x 1卷积层
        self.p1_1 = nn.Conv2d(in_c, c1, kernel_size=1)
        # 线路2，1 x 1卷积层后接3 x 3卷积层
        self.p2_1 = nn.Conv2d(in_c, c2[0], kernel_size=1)
        self.p2_2 = nn.Conv2d(c2[0], c2[1], kernel_size=3, padding=1)
        # 线路3，1 x 1卷积层后接5 x 5卷积层
        self.p3_1 = nn.Conv2d(in_c, c3[0], kernel_size=1)
        self.p3_2 = nn.Conv2d(c3[0], c3[1], kernel_size=5, padding=2)
        # 线路4，3 x 3最大池化层后接1 x 1卷积层
        self.p4_1 = nn.MaxPool2d(kernel_size=3, stride=1, padding=1)
        self.p4_2 = nn.Conv2d(in_c, c4, kernel_size=1)

    def forward(self, x):
        p1 = F.relu(self.p1_1(x))
        p2 = F.relu(self.p2_2(F.relu(self.p2_1(x))))
        p3 = F.relu(self.p3_2(F.relu(self.p3_1(x))))
        p4 = F.relu(self.p4_2(self.p4_1(x)))
        return torch.cat((p1, p2, p3, p4), dim=1)  # 在通道维上连结输出
```

```
b1 = nn.Sequential(nn.Conv2d(1, 64, kernel_size=7, stride=2, padding=3),
                   nn.ReLU(),
                   nn.MaxPool2d(kernel_size=3, stride=2, padding=1))

b2 = nn.Sequential(nn.Conv2d(64, 64, kernel_size=1),
                   nn.Conv2d(64, 192, kernel_size=3, padding=1),
                   nn.MaxPool2d(kernel_size=3, stride=2, padding=1))

b3 = nn.Sequential(Inception(192, 64, (96, 128), (16, 32), 32),
                   Inception(256, 128, (128, 192), (32, 96), 64),
                   nn.MaxPool2d(kernel_size=3, stride=2, padding=1))

b4 = nn.Sequential(Inception(480, 192, (96, 208), (16, 48), 64),
                   Inception(512, 160, (112, 224), (24, 64), 64),
                   Inception(512, 128, (128, 256), (24, 64), 64),
                   Inception(512, 112, (144, 288), (32, 64), 64),
                   Inception(528, 256, (160, 320), (32, 128), 128),
                   nn.MaxPool2d(kernel_size=3, stride=2, padding=1))

b5 = nn.Sequential(Inception(832, 256, (160, 320), (32, 128), 128),
                   Inception(832, 384, (192, 384), (48, 128), 128),
                   d2l.GlobalAvgPool2d())

net = nn.Sequential(b1, b2, b3, b4, b5,
                    d2l.FlattenLayer(), nn.Linear(1024, 10))

net = nn.Sequential(b1, b2, b3, b4, b5, d2l.FlattenLayer(), nn.Linear(1024, 10))
```

PREVIOUS
PYTORCH系列之机器翻译 | 注意力机制 | TRANSFORMER
(/2020/02/14/%E6%9C%BA%E5%99%A8%E7%BF%BB%E8%AF%91-%E6%B3%A8%E6%84%8F%E5%8A%9B%E6%9C%BA%E5%88%B6-TRANSFORMER/)

NEXT
PYTORCH系列之批量归一化和残差网络 | 凸优化 | 梯度下降
**(/2020/02/16/%E6%89%B9%E9%87%8F%E5%BD%92%E4%B8%80%E5%8C%96%E4%B8%8E%E9%A
%E5%87%B8%E4%BC%98%E5%8C%96/)**