Task 05

1. 卷积神经网络基础

1.1 二维卷积层

- 二维互相关运算 (**深度学习中的卷积运算实际上是互相关运算**)

```
#这里的步长为1 没有padding
import torch
def corr2d(X,k):
    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。

```
class Conv2D(nn.Module):
    def __init__(self,kernel_size):
        #调用父类的初始化方法  kernel_size 长度为2的元组 (高，宽)
        super(Conv2d, self).__init__()
        #定义两个成员变量
        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
```

- 边缘检测的卷积核训练

> 学习一个卷积层，用来检测颜色边缘 （代码略）

小结:

> 1、二维卷积层的核心心计算是二二维互相关运算。在最简单的形式下，它对二二维输入入数据和卷积核做互相关运算然后加上偏差。
> 2、我们可以设计卷积核来检测图像中的边缘。
> 3、我们可以通过数据来学习卷积核。

习题

- 假如你用全连接层处理一张256 X 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：196609000。（图像展平后长度为3×256×256，权重参数和偏置参数的数量是3×256×256×1000+1000=196609000）
- 假如你用全连接层处理一张 256256×256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： 280。（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。）
- 池化层有参与模型的正向计算，同样也会参与反向传播；

- 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；

2. LeNet

Convolutional Neural Networks

优势:

> 卷积层保留输入形状。
>
> 卷积层通过滑动窗口将同一卷积核与不同位置的输入重复计算，从而避免参数尺寸过大。

LeNet模型

LeNet分为卷积层块和全连接层块两个部分。下面我们分别介绍这两个模块。

- 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5x5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。

GPU

```
def try_gpu():
    """If GPU is available, return torch.device as cuda:0; else return
torch.device as cpu."""
    if torch.cuda.is_available():
        device = torch.device('cuda:0')
    else:
        device = torch.device('cpu')
    return device

device = try_gpu()
device
```

pytorch维度

> pytorch中的tensor维度可以通过第一个数前面的中括号数量来判断，有几个中括号维度就是多少。维度标号从0开始。过一个括号是一维，同一维度内元素用逗号隔开。

卷积神经网络进阶

卷积:

> 卷积的输入（batch_size, input_channels, input_height, input_width）
>
> (定义时) 卷积层（input_channels, output_channels, kernels, ...）
>
> 卷积变量　（训练学习到的）

> weight(`tensor`) - 卷积的权重，大小是(`out_channels`, `in_channels`, `kernel_height`, `kernel_height`);
>
> bias(`tensor`) - 卷积的偏置系数，大小是（`out_channel`）
>
> 卷积层 输出结果 (batch_size(N), out_channels, height, width)（输出的结果）

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：**

> 1. 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。
> 2. 将sigmoid激活函数改成了更加简单的ReLU激活函数。
> 3. 用Dropout来控制全连接层的模型复杂度。
> 4. 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。

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 x 3的卷积层,接上一个步幅为2、窗口形状为 2 x 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。

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(N)元素求平均并直接用于分类。

1×1卷积核作用：
1.放缩通道数：通过控制卷积核的数量达到通道数的放缩。
2.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。
3.计算参数少

(全连接层会被取代)

- 放缩通道

> - 1×1 没有改变 height 和 width，改变通道的第一个最直观的结果，就是可以将原本的数据量进行增加或者减少，改变的只是 height × width × channels 中的 channels 维度大小。
> - ResNet同样也利用了1×1卷积，并且是在3×3卷积层的前后都使用了，不仅进行了降维，还进行了升维，使得卷积层的输入和输出的通道数都减小，参数数量进一步减少

- 为什么NiN块里要有两个1 × 1卷积层？去除其中的一个，观察并分析实验现象

实验结果对比：

上面的图是NiN中只有一个1×1 卷积层（a），下面的图有两个1×1卷积层(b)

```
14
15 batch_size = 128
16 # 如出现“out of memory”的报错信息，可减小batch_size或resize
17 #train_iter, test_iter = d2l.load_data_fashion_mnist(batch_size, resize=224)
18
19 lr, num_epochs = 0.002, 5
20 optimizer = torch.optim.Adam(net.parameters(), lr=lr)
21 d2l.train_ch5(net, train_iter, test_iter, batch_size, optimizer, device, num_epochs)
```

```
training on  cuda:0
epoch 1, loss 1.5870, train acc 0.402, test acc 0.419, time 68.1 sec
epoch 2, loss 1.4821, train acc 0.419, test acc 0.420, time 68.0 sec
epoch 3, loss 1.4560, train acc 0.424, test acc 0.417, time 67.9 sec
epoch 4, loss 1.4471, train acc 0.426, test acc 0.421, time 68.0 sec
epoch 5, loss 1.4435, train acc 0.420, test acc 0.428, time 68.0 sec
```

In [26]:

```
1 batch_size = 128
2 # 如出现“out of memory”的报错信息，可减小batch_size或resize
3 #train_iter, test_iter = d2l.load_data_fashion_mnist(batch_size, resize=224)
4
5 lr, num_epochs = 0.002, 5
6 optimizer = torch.optim.Adam(net.parameters(), lr=lr)
7 d2l.train_ch5(net, train_iter, test_iter, batch_size, optimizer, device, num_epochs)
```

```
training on  cuda:0
epoch 1, loss 1.3789, train acc 0.479, test acc 0.668, time 82.4 sec
epoch 2, loss 0.8426, train acc 0.691, test acc 0.708, time 82.3 sec
epoch 3, loss 0.7522, train acc 0.726, test acc 0.737, time 82.4 sec
epoch 4, loss 0.7127, train acc 0.742, test acc 0.746, time 82.4 sec
epoch 5, loss 0.6907, train acc 0.750, test acc 0.752, time 82.2 sec
```

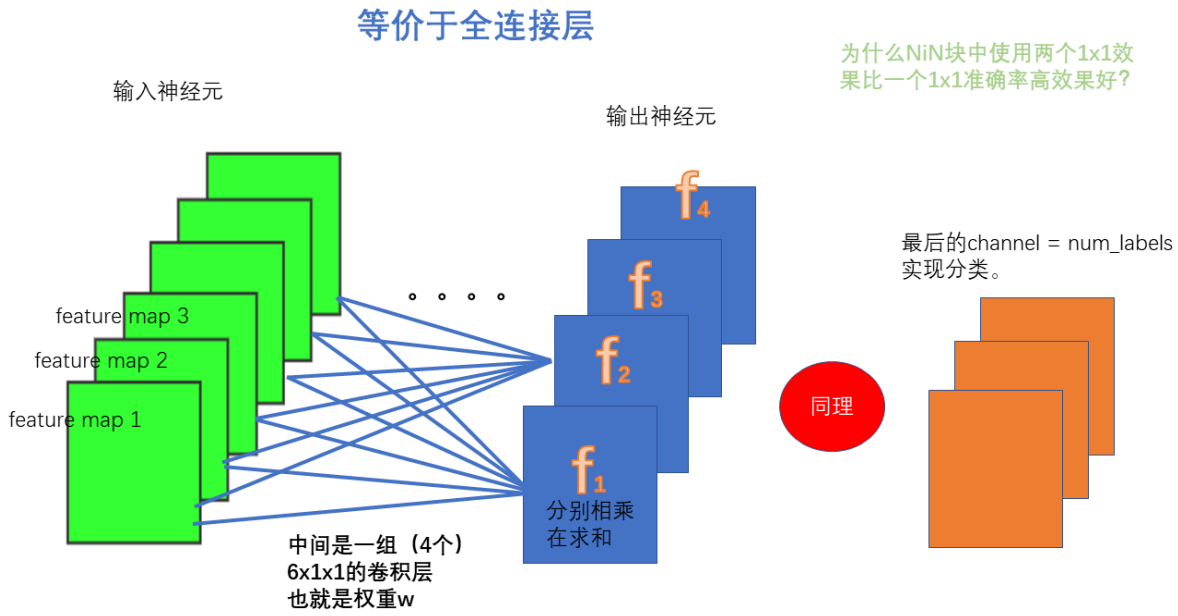
可以发现虽然(a)运行时间缩短，模型复杂度降低，但是准确率也明显低于(b)。

解释：

- 1x1卷积可以**实现多个feature map的线性组合，实现跨通道的信息整合**的功效

> CNN里的卷积大都是多通道的feature map和多通道的卷积核之间的操作（输入的多通道的feature map和一组卷积核做卷积求和得到一个输出的feature map），如果使用一组1x1的卷积核，这个操作实现的就是多个feature map的线性组合，一个1x1的卷积核实质就是一个数，一组（组数必然等于输入feature map的通道数）就是很多个数，可以实现feature map在通道个数上的变化。

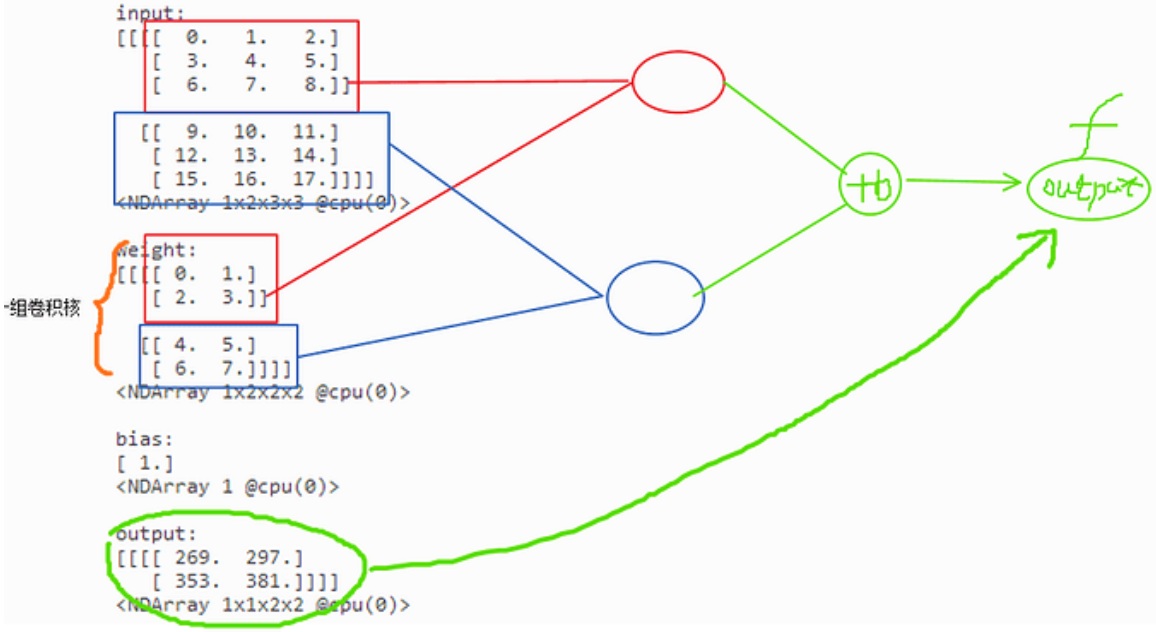
简单示意图：

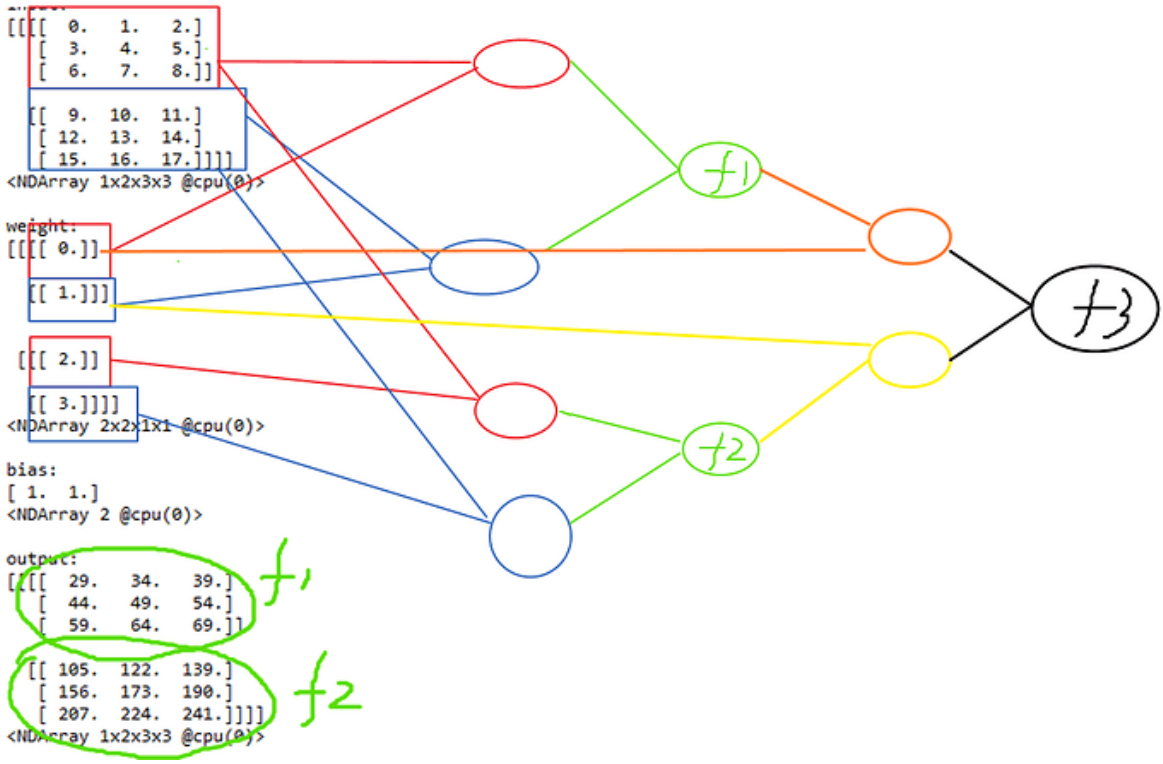


官网讨论区：（上图是1个 1x1卷积，下图是两个）

In [3]:
```
w = nd.arange(8).reshape((1,2,2,2))
data = nd.arange(18).reshape((1,2,3,3))

out = nd.Convolution(data, w, b, kernel=w.shape[2:], num_filter=w.shape[0])

print('input:', data, '\n\nweight:', w, '\n\nbias:', b, '\n\noutput:', out)
```

```
input:
[[[[  0.   1.   2.]
   [  3.   4.   5.]
   [  6.   7.   8.]]

  [[  9.  10.  11.]
   [ 12.  13.  14.]
   [ 15.  16.  17.]]]]
<NDArray 1x2x3x3 @cpu(0)>

weight:
[[[[ 0.  1.]
   [ 2.  3.]]

  [[ 4.  5.]
   [ 6.  7.]]]]
<NDArray 1x2x2x2 @cpu(0)>

bias:
[ 1.]
<NDArray 1 @cpu(0)>

output:
[[[[ 269.  297.]
   [ 353.  381.]]]]
<NDArray 1x1x2x2 @cpu(0)>
```






```
[[[[  0.   1.   2.]
   [  3.   4.   5.]
   [  6.   7.   8.]]

  [[  9.  10.  11.]
   [ 12.  13.  14.]
   [ 15.  16.  17.]]]]
<NDArray 1x2x3x3 @cpu(0)>

weight:
[[[[ 0.]]

  [[ 1.]]]


 [[[ 2.]]

  [[ 3.]]]]
<NDArray 2x2x1x1 @cpu(0)>

bias:
[ 1.  1.]
<NDArray 2 @cpu(0)>

output:
[[[[  29.   34.   39.]
   [  44.   49.   54.]
   [  59.   64.   69.]]

  [[ 105.  122.  139.]
   [ 156.  173.  190.]
   [ 207.  224.  241.]]]]
<NDArray 1x2x3x3 @cpu(0)>
```