Notebook credit: Based on the original D2L notebook here.

Multiple Input and Multiple Output Channels

While we have described the multiple channels that comprise each image (e.g., color images have the standard RGB channels to indicate the amount of red, green and blue) and convolutional layers for multiple channels before, until now, we simplified all of our numerical examples by working with just a single input and a single output channel. This has allowed us to think of our inputs, convolution kernels, and outputs each as two-dimensional tensors.

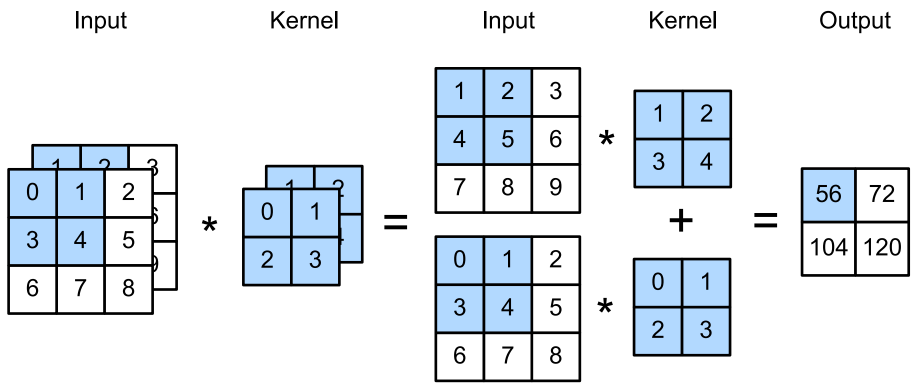
When we add channels into the mix, our inputs and hidden representations both become three-dimensional tensors. For example, each RGB input image has shape $3 \times h \times w$. We refer to this axis, with a size of 3, as the *channel* dimension. In this section, we will take a deeper look at convolution kernels with multiple input and multiple output channels.

Multiple Input Channels

When the input data contain multiple channels, we need to construct a convolution kernel with the same number of input channels as the input data, so that it can perform cross-correlation with the input data. Assuming that the number of channels for the input data is c_i, the number of input channels of the convolution kernel also needs to be c_i. If our convolution kernel's window shape is $k_h \times k_w$, then when $c_i = 1$, we can think of our convolution kernel as just a two-dimensional tensor of shape $k_h \times k_w$.

However, when $c_i > 1$, we need a kernel that contains a tensor of shape $k_h \times k_w$ for *every* input channel. Concatenating these c_i tensors together yields a convolution kernel of shape $c_i \times k_h \times k_w$. Since the input and convolution kernel each have c_i channels, we can perform a cross-correlation operation on the two-dimensional tensor of the input and the two-dimensional tensor of the convolution kernel for each channel, adding the c_i results together (summing over the channels) to yield a two-dimensional tensor. This is the result of a two-dimensional cross-correlation between a multi-channel input and a multi-input-channel convolution kernel.

In the figure below, we demonstrate an example of a two-dimensional cross-correlation with two input channels. The shaded portions are the first output element as well as the input and kernel tensor elements used for the output computation: $(1 \times 1 + 2 \times 2 + 4 \times 3 + 5 \times 4) + (0 \times 0 + 1 \times 1 + 3 \times 2 + 4 \times 3) = 56.$



To make sure we really understand what is going on here, we can (**implement cross-correlation operations with multiple input channels**) ourselves. Notice that all we are doing is performing one cross-correlation operation per channel and then adding up the results.

在 `corr2d` 的基础上 implement `corr2d_milti_in`: 取多 channels 的 convolution output 的 sum

```
import tensorflow as tf

def corr2d(X, K):
    """Compute 2D cross-correlation."""
    h, w = K.shape
    Y = tf.Variable(tf.zeros((X.shape[0] - h + 1, X.shape[1] - w + 1)))
    for i in range(Y.shape[0]):
        for j in range(Y.shape[1]):
            Y[i, j].assign(tf.reduce_sum(
                X[i: i + h, j: j + w] * K))
    return Y

def corr2d_multi_in(X, K):
    # First, iterate through the 0th dimension (channel dimension) of `X` and
    # `K`. Then, add them together
    return tf.reduce_sum([corr2d(x, k) for x, k in zip(X, K)], axis=0)
```

We can construct the input tensor `X` and the kernel tensor `K` corresponding to the values in the figure above to (**validate the output**) of the cross-correlation operation.

```
X = tf.constant([[[0.0, 1.0, 2.0], [3.0, 4.0, 5.0], [6.0, 7.0, 8.0]],
               [[1.0, 2.0, 3.0], [4.0, 5.0, 6.0], [7.0, 8.0, 9.0]]])
K = tf.constant([[[0.0, 1.0], [2.0, 3.0]], [[1.0, 2.0], [3.0, 4.0]]])

corr2d_multi_in(X, K).numpy()
```

```
    array([[ 56.,  72.],
           [104., 120.]], dtype=float32)
```

Multiple Output Channels

无论 input channels 的数量是多少，到目前为止，我们始终只有一个 output channel. 然而事实证明，在每个 layer 拥有多个 channels 是至关重要的。在最流行的 NN architectures 中，我们实际上是随着layers 的增加而增加 channel dimensions，通常是通过 downsamplin(降低采样率), 以 spatial resolution(空间分辨率)换取更大的*channel depth*(通道深度). 直观地说，你可以把每个通道看作是对一些不同特征的响应. 现实要比对这一直觉的 naive interpretations 更复杂一些，因为现实中的 representations 并不是独立学习的，而是 optimized to be jointly useful. 因此，可能不是单个通道学习为了 edge detector, 而是 channel space 中的某个 direction 对应于 detecting edges.

用 c_i and c_o 分别表示 input channels 和 output channels 的数量. input and output channels. 用 k_h 和 k_w 表示 kernel 的 height 和 width.

为了获取 multiple channels 的 output, 我们可以**为每个 output channel 创建一个形状为 $c_i \times k_h \times k_w$ 的 kernel tensor.** 然后我们 concatenate them on the output channel dimension, so that the shape of the convolution kernel is $c_o \times c_i \times k_h \times k_w$.

In cross-correlation operations, the result on each output channel is calculated from the convolution kernel corresponding to that output channel and takes input from all channels in the input tensor. 在 cross-correlation operations中, 每个 output channe 的结果都是从 input tensor 的所有 channels 中获取 input, 并通过与其相对应的 convolution kernel 计算得出的.

简单而言: 一个 kernel tensor 就对应一个 output channel. 如果有多个 input channel, 那么每个 output channel 都是所有 input channel 在同一个 kernel tensor 下作用的和.

We implement a cross-correlation function to [**calculate the output of multiple channels**] as shown below.

```
def corr2d_multi_in_out(X, K):
    # Iterate through the 0th dimension of `K`, and each time, perform
    # cross-correlation operations with input `X`. All of the results are
    # stacked together
    return tf.stack([corr2d_multi_in(X, k) for k in K])
```

We construct a convolution kernel with 3 output channels by concatenating the kernel tensor `K` with `K+1` (plus one for each element in `K`) and `K+2`.

```
K = tf.stack((K, K + 1, K + 2))
K.shape
```

```
    TensorShape([3, 2, 2, 2])
```

Below, we perform cross-correlation operations on the input tensor `X` with the kernel tensor `K`. Now the output contains 3 channels. The result of the first channel is consistent with the result of the previous input tensor `X` and the multi-input channel, single-output channel kernel.

```
corr2d_multi_in_out(X, K)

    <tf.Tensor: shape=(3, 2, 2), dtype=float32, numpy=
    array([[[ 56.,  72.],
            [104., 120.]],

           [[ 76., 100.],
            [148., 172.]],

           [[ 96., 128.],
            [192., 224.]]], dtype=float32)>
```

1×1 Convolutional Layer

At first, a [1×1 **convolution**], i.e., $k_h = k_w = 1$, does not seem to make much sense. After all, a convolution correlates adjacent pixels. A 1×1 convolution obviously does not. Nonetheless, they are popular operations that are sometimes included in the designs of complex deep networks. Let us see in some detail what it actually does.

我们可以梳理一下 1×1 convolution kernel 的特征:

- 首先 1×1 convolution kernel 是没有普通的 convolutional layers 的识别一个 entry 的上下左右的 entries 之间的 interactions 的模式的功能的.
- 其次 1×1 convolution kernel 的输入和输出的单个 channel 的大小总是一样的. (by the formula).

实际上, the only computation of the 1×1 convolution occurs on the channel dimension.

You could think of the 1×1 convolutional layer as constituting a fully-connected layer applied at every single pixel location to transform the c_i corresponding input values into c_o output values.

在卷积神经网络中, 1×1的卷积层对于所有的位置都只应用同一个 weight.

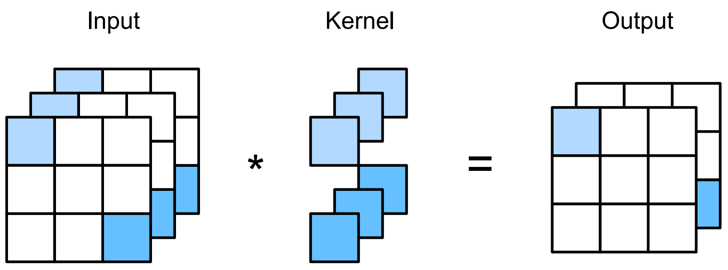
因而 1×1 convolution layer 需要 $c_o \times c_i$ 个 weights 加上 bias (如果每个 output channel 都用一个 bias, 则一共是 $(c_o + 1) \times (c_i + 1)$ 个参数).

- c_i: input channels 的数量.
- c_o: output channels 的数量.

这样理解更好: 有 n 个 output channels 就有 n 个 kernel tensors; 而有 m 个 input channels, 每个 kernel tensors 中有 m 个 1x1 小块. 于是有 $n \times m$ 个 weights. 在每个 kernel tensor 中, m 个 weights

代表了这个 kernel tensor 所对应的 output channel 中, 每个 input channel 所占的 weights. 于是我们通过一个线性变换, 把 n 个 channels 变成了 m 个 channels.

下图展示了使用 1×1 convolution kernel 的 cross-correlation computation with 3 input channels and 2 output channels. 这里我们有 3 个 input channels 和 2 个 output channels, 于是有 2 个 kernel tensors, 每个 kernel tensor 有 3 个 subtensor, 因而一共有 6 个 1x1 的小 tensor.



1×1 Convolutional Layer 的作用

在卷积神经网络中，不同层的输出可能会有不同数量的 channels(=feature maps 的数量).

有时候, 为了进一步处理或简化网络架构, 我们可能需要增加或减少这些通道的数量. 使用1×1的卷积层可以实现这一点。通过指定1×1卷积层的输出通道数, 我们可以控制经过该层之后的通道数. 例如, 如果上一层产生了256个通道的输出, 但我们希望下一层的输入只有64个通道, 我们可以使用一个输出通道数为64的1×1卷积层来减少通道数.

(其实就是: 一个 kernel tensor 中的每个小 tensor 代表给一个 input channel 设置的权重, 这样每个 output channel 就是所有 input tensors 的一个线性组合; 于是, n 个 kernel tensors 就代表了 n 个 output channels, 每个 output channels 都是所有 input channels 的一个线性组合. 于是, 我们就把所有 input channel 给 output 成了 n 个新的 output channels, 从而改变了下一层的 input neurons 的数量.)

```
def corr2d_multi_in_out_1x1(X, K):
    c_i, h, w = X.shape
    c_o = K.shape[0]
    X = tf.reshape(X, (c_i, h * w))
    K = tf.reshape(K, (c_o, c_i))
    # Matrix multiplication in the fully-connected layer
    Y = tf.matmul(K, X)
    return tf.reshape(Y, (c_o, h, w))
```

When performing 1×1 convolution, the above function is equivalent to the previously implemented cross-correlation function `corr2d_multi_in_out`. Let us check this with some sample data.

```
X = tf.random.normal((3, 3, 3))
K = tf.random.normal((2, 3, 1, 1))


Y1 = corr2d_multi_in_out_1x1(X, K)
Y2 = corr2d_multi_in_out(X, K)
assert float(tf.reduce_sum(tf.abs(Y1 - Y2))) < 1e-6
```

Summary

- 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.