```
!pip install d2l==1.0.3
```

Show hidden output

COSE474-2024F: Deep Learning

7.4. Multiple Input and Multiple Output Channels

While we 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 in Section 7.1.4, until now, we simplified all of our numerical examples by working with just a single input and a single output channel. This 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. The notion of channels is as old as CNNs themselves: for instance LeNet-5 (LeCun et al., 1995) uses them. In this section, we will take a deeper look at convolution kernels with multiple input and multiple output channels.

```
import torch
from d2l import torch as d2l
```

7.4.1. Multiple Input Channels

When the input data contains 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 $k_\mathrm{h} \times k_\mathrm{w}$, then, when $c_\mathrm{i} = 1$, we can think of our convolution kernel as just a two-dimensional tensor of shape $k_\mathrm{h} \times k_\mathrm{w}$.

However, when $c_\mathrm{i} > 1$, we need a kernel that contains a tensor of shape $k_\mathrm{h} \times k_\mathrm{w}$ for *every* input channel. Concatenating these c_i tensors together yields a convolution kernel of shape $c_\mathrm{i} \times k_\mathrm{h} \times k_\mathrm{w}$. 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.

Fig. 7.4.1 provides 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:
$(1 \times 1 + 2 \times 2 + 4 \times 3 + 5 \times 4) + (0 \times 0 + 1 \times 1 + 3 \times 2 + 4 \times 3) = 56.$

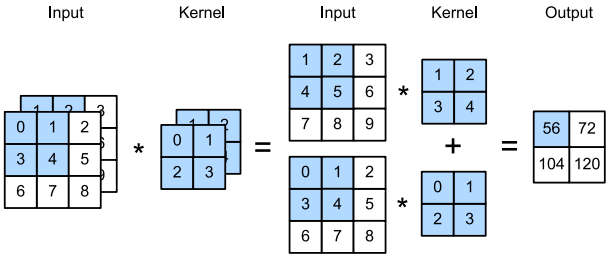

Fig. 7.4.1 Cross-correlation computation with two input channels.

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 a cross-correlation operation per channel and then adding up the results.

```
def corr2d_multi_in(X, K):
    # Iterate through the 0th dimension (channel) of K first, then add them up
    return sum(d2l.corr2d(x, k) for x, k in zip(X, K))
```

We can construct the input tensor X and the kernel tensor K corresponding to the values in Fig. 7.4.1 to validate the output of the cross-correlation operation.

```
X = torch.tensor([[[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 = torch.tensor([[[0.0, 1.0], [2.0, 3.0]], [[1.0, 2.0], [3.0, 4.0]]])

corr2d_multi_in(X, K)
```

```
tensor([[ 56.,  72.],
        [104., 120.]])
```

7.4.2. Multiple Output Channels

Regardless of the number of input channels, so far we always ended up with one output channel. However, as we discussed in Section 7.1.4, it turns out to be essential to have multiple channels at each layer. In the most popular neural network architectures, we actually increase the channel dimension as we go deeper in the neural network, typically downsampling to trade off spatial resolution for greater channel depth. Intuitively, you could think of each channel as responding to a different set of features. The reality is a bit more complicated than this. A naive interpretation would suggest that representations are learned independently per pixel or per channel. Instead, channels are optimized to be jointly useful. This means that rather than mapping a single channel to an edge detector, it may simply mean that some direction in channel space corresponds to detecting edges.

Denote by c_i and c_o the number of input and output channels, respectively, and by k_h and k_w the height and width of the kernel. To get an output with multiple channels, we can create a kernel tensor of shape $c_\mathrm{i} \times k_\mathrm{h} \times k_\mathrm{w}$ for *every* output channel. We concatenate them on the output channel dimension, so that the shape of the convolution kernel is $c_\mathrm{o} \times c_\mathrm{i} \times k_\mathrm{h} \times k_\mathrm{w}$. 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.

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 torch.stack([corr2d_multi_in(X, k) for k in K], 0)
```

We construct a trivial convolution kernel with three output channels by concatenating the kernel tensor for K with K+1 and K+2.

```
K = torch.stack((K, K + 1, K + 2), 0)
K.shape
```

```
torch.Size([3, 2, 2, 2])
```

Below, we perform cross-correlation operations on the input tensor X with the kernel tensor K. Now the output contains three 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)
```

```
tensor([[[ 56.,  72.],
         [104., 120.]],

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

        [[ 96., 128.],
         [192., 224.]]])
```

7.4.3. 1×1 Convolutional Layer

At first, a [1×1 **convolution**], i.e., $k_\mathrm{h} = k_\mathrm{w} = 1$, 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 (Lin et al., 2013, Szegedy et al., 2017). Let's see in some detail what it actually does.

Because the minimum window is used, the 1×1 convolution loses the ability of larger convolutional layers to recognize patterns consisting of interactions among adjacent elements in the height and width dimensions. The only computation of the 1×1 convolution occurs on the

channel dimension.

Fig. 7.4.2 shows the cross-correlation computation using the 1×1 convolution kernel with 3 input channels and 2 output channels. Note that the inputs and outputs have the same height and width. Each element in the output is derived from a linear combination of elements *at the same position* in the input image. 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. Because this is still a convolutional layer, the weights are tied across pixel location. Thus the 1×1 convolutional layer requires $c_\mathrm{o} \times c_\mathrm{i}$ weights (plus the bias). Also note that convolutional layers are typically followed by nonlinearities. This ensures that 1×1 convolutions cannot simply be folded into other convolutions.

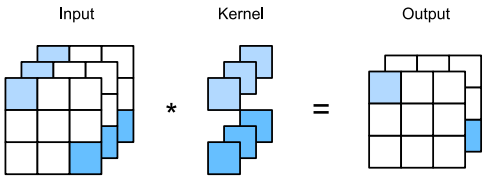

Fig. 7.4.2 The cross-correlation computation uses the $[1 \times 1]$ convolution kernel with three input channels and two output channels. The input and output have the same height and width.

Let's check whether this works in practice: we implement a 1×1 convolution using a fully connected layer. The only thing is that we need to make some adjustments to the data shape before and after the matrix multiplication.

```
def corr2d_multi_in_out_1x1(X, K):
    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))
    # Matrix multiplication in the fully connected layer
    Y = torch.matmul(K, X)
    return Y.reshape((c_o, h, w))
```

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

```
X = torch.normal(0, 1, (3, 3, 3))
K = torch.normal(0, 1, (2, 3, 1, 1))
Y1 = corr2d_multi_in_out_1x1(X, K)
Y2 = corr2d_multi_in_out(X, K)
assert float(torch.abs(Y1 - Y2).sum()) < 1e-6
```

7.4.4. Discussion

- Channels allow us to combine the best of both worlds
- MLPs that allow for significant nonlinearities and convolutions that allow for localized analysis of features.
- Channels allow the CNN to reason with multiple features, such as edge and shape detectors at the same time.
- Offer a practical trade-off between the drastic parameter reduction arising from translation invariance and locality, and the need for expressive and diverse models in computer vision.